Echo flag locaties.

```
Loc. Data
0316 7F F1 19  CLEAR ECHO FLAG(SET ECHO ON)        F119 =  A07F
033C 7F F1 19  IDEM                                        A00C
0425 73 F1 19  COM ECHO
0643 7F F1 19  CLEAR
0895 7D F1 19  TST ECHO
08DA 7F F1 19  CLEAR
```

Als vanaf locatie 0895 vijf NOP's ingevuld worden
en op loc. F119 zit ROM of geen geheugen dan
kan de rest ongewijzigd blijven.

==

BASIC INCH SUBROUTINE

WORDT AANGEROEPEN VANAF LOC 041A = 18 F3
GEEN JMP MAAR JSR INSTRUKTIE GEBRUIKEN.

=== KAN EVENTUEEL NAAR AUTO-LINE WIJZEN ===

==

BASIC OUTCH SUBROUTINE

AANROEP VANAF LOC. 08AB = A07∅ A07∅ = 847F
WEL MET JMP INSTRUKTIE (7E) 7E E1D1

=== ALLE I/O WERKT MET ACCU A ===
=== ACCU B MAG BIJ INCH NIET VERLOREN GAAN ===

==

BASIC BREAK TEST
het volgende programma heb ik gebruikt om
het keyboard te testen.

```
0620 36        BR.TEST  PSH A
0621 B6 E4 00           LDA A   KEYBOARD
0624 32                 PUL A
0625 2B F0              BMI     0617      zolang linkse bit =1 dan RTS(op loc.0617)
0627 B6 E4 00           LDA A   KEYBOARD  als bit 7=0 dan is er een toets ingedrukt
062A 81 0B              CMP A £ 0B        is het de BREAK-toets?
062C 01                 NOP
```

```
0620  36         PSH A
  21  B6 F000    LDA A  $ F000
  24  47         ASR A
  25  32         PUL A
  26  24 EF      BCC
  28  BD 041A    JSR    $ 041A
  2B  81 03      CMP A  # $∅3
  2D  0D         SEC
  2E  26 33      BNE
  30  07         TPA
  31  DE C8      LDX    $ C8
```

RMINAL CONTROL

~~000a-terminal-~~

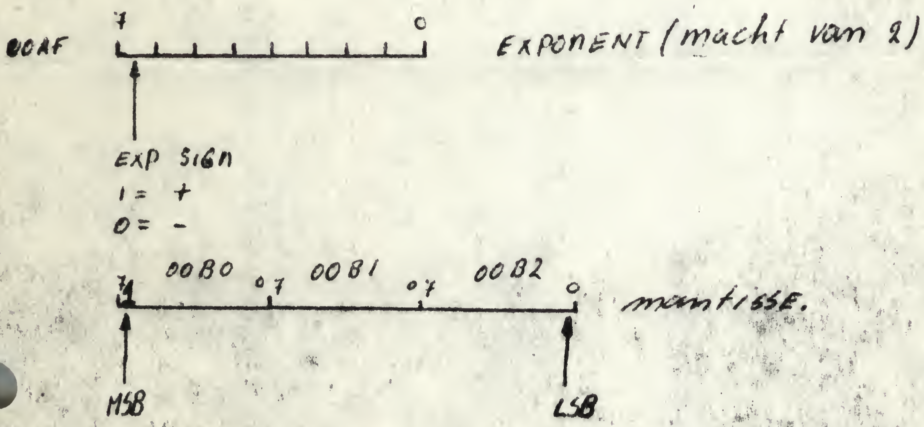
000A aantal NULL characters bij LF/CR = 3
000B cursor position
000C terminal breedte
000D laatste kolom
0010-0056 input buffer
03F7 backspace character = ∅8
03E0 reply on backspace input = ∅∅
03FB carriage return(reply=LF/CR)
03FF line cancel character

wanneer een HT (TAB) als skip character
gebruikt moet worden(wel een plaats vooruit en niet modificeren in de buffer)
dan moeten deze patches gemaakt worden.
03F2 81 <u>09</u> 27 12

USER SUBROUTINE

aanroep adres staat op 011F/0120

getal te vinden van 00AF-00B3



van 00B3 WORDT alleen het MSB gebruikt
als teken-bit v/d mantisse.

Caminide

£ character i.p.v #, # zit niet op type machine.

ALTAIR BASIC AUTO-LINE AANPASSEN

```
          E1 AC
18F3 BD F8 9B AUTO     JSR       INCH    basic inch moet hier heen wijzen
18F6 C1 01             CMP B £ 01        accu B=input buffer-pointer 01=eerste positie
18F8 27 01             BEQ       18FB
18FA 39                RTS
18FB 81 0E             CMP A £ 0E        0E=special character for aoto-line
18FD 26 FB             BNE       18FA
18FF D6 8F             LDA B     008F    008F&008E regelnummer van vorige regel
1901 CB 0A             ADD B £ 0A        regelnummer is binair
1903 96 8E             LDA A     008E
1905 89 00             ADC A £ 00
1907 BD 15 E0          JSR       15F0    15e0 sbroutine output a&b decimaal(aanwezig in basic)
190A CE 00 10          LDX     £ 0010    de ascii characters van de output zijn terug te
190D C6 01             LDA B £ 01        vinden vanaf lokatie 0101 afgesloten door 00
190F A6 F1             LDA A X F1
1911 27 06             BEQ       1919
1913 A7 00             STA A X 00        block transfer naar 0010 (input buffer)
1915 5C                INC B
1916 08                INX
1917 20 F6             BRA       190F
1919 7E 08 90          JMP       0890    subr spatie (+RTS)
```

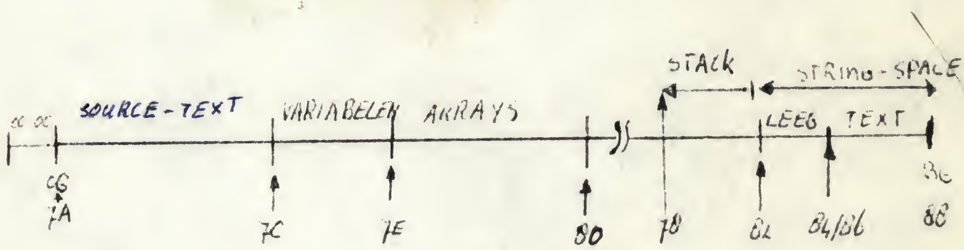
```
=========================================
=                                       =
= DE ondergrens van het gebruikers-     =
= geheugen is nu 191E geworden          =
=                                       =
=========================================
```

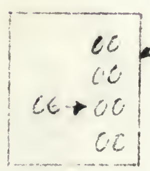
MEMORY SIZE

OG=ondergrens
BG=bovengrens

0078=0082
007A=OG
007C=OG+2
007E=OG+2
0080=OG+2
0082=BG-stringspace
0084=BG
0086=BG+1
0088=BG
0096=OG-1 } pointers voor:
0098=OG-1 }

DATA
READ
RESTORE





bij OPSTARTEN

END = 191B

STRING SPACE TE VERANDEREN DOOR:
CLEAR XX
nieuwe STRING SPACE.

INCH 1900
OUTCH 190C

Loc	Data	Function
0316	7F F119	clear no echo locatie = echo on
033C	7F F119	idem
041A	BD FF0D	TO INCH. SUBROUTINE
0425	73 F119	Set no echo flag = NO ECHO
0621	B6 F000 ACIA	test for break.
	47	look for char
	32	
	24 F0	
	BD 041A	TO INCH
	81 03	IS IT ↑C.
0643	7F F119	clear no echo flag : echo
0895	7D F119	test echo flag.
08AB	7E FEFD	TO subr outch.
08BA	7F F119	clear no echo flag : echo

00CB = 3A – 6A
0096 = 39 – 69

Start = 000Ø HOT OR COLD.

F119 is RAM loc for: echo/noecho flag

control D = toggle for echo|no echo

Control O = backspace

Control X = line cancel

Control C = Break.

FF0D subroutine adress input char

FEFD " " outp. char

000C is loc for terminal width (hex)

Start of buffer 1ABF

End of buffer 3AFF = ≃ 15K.

PATCHES to be made before RUN

program = 0000 – 1A[illegible]F STACK P. 0092

Loc	Data	Function.
x 0316	7F ~~F119~~ → A07F	clear no echo location = ECHO ON
x 033C	7F F119	idem
X 041A	BD FF0D E1AC	TO INCH SUBROUTINE
x 0425	73 F119	Set no echo flag - NO ECHO
0621	B6 E000 ACIA	test for break. look for char.
	47	
	32	
	24 ~~F6~~ EF	
	30 0421A	TO INCH
	81 03	IS IT ↑C.
0643	7F F119	clear no echo flag : echo
x 0895	7D F119	test echo flag
x 08AB	7E FEFD / A070	TO subr outch.
x 08DA	7F F119	clear no echo flag : echo

Start = 0000 HOT OR COLD.

F119 is RAM loc for: echo / no echo flag

control D = toggle for echo | no echo

Control O = backspace

Control X = line cancel

Control C = Break.

FF0D subroutine adress input char

FEFD " " " outp. char.

000C is loc for terminal width (hex)

Start of buffer 1ABF

End of buffer 3AFF = ≈ 15k.

PATCHES to be made before RUN

Program = 0000 - 1ABF

0082 - 3ACC → A060 — STACK P
0084 - 3AFF → 2FFF } END OF MEMORY.
0088 - 3AFF → 2FFF
0078 - 3AC5 → A058 — SP

Aø7ø 84 7F
7E E101

ø621 B6 8øø4
01
01
2B EF

0092 A06F

0094
0093

The argument is obtained for use in the user function by calling the routine whose address is given in locations Ø115 and Ø116 (hexadecimal). Therefore, the instructions

```
LDX $Ø115
JSR X
```

cause the argument to be converted to a signed two byte integer with the high order byte stored location $174_{(10)}$ and the low order byte stored in $175_{(10)}$. 00B1 + 00B2

The result of the function is returned to BASIC by storing the high order byte in accumulator A, the low order byte in accumulator B, and calling the routine whose address is given in locations Ø117 and Ø118 (hexadecimal).

For example, the instructions

```
CLR A
LDA B #3
LDX $Ø117
JSR X
```

will return a value of 3 to BASIC. Program control is returned to BASIC by executing an RTS instruction.

Example USR Function

The USR function described below generates a program delay of 1 second times the argument. The function assumes the argument is between 1 and $256_{(10)}$. The value returned to BASIC is always zero. It is assumed the user answered the memory size question with 16383.

```
00001                     NAM      USRFN
00002 4000                ORG      $4000
00003 4000 FE 0115        LDX      $0115    GET LOW BYTE OF ARG
00004 4003 AD 00          JSR      X        INTØ B
00005 4005 D6 AF          LDA B    175      WE ASSUME HIGH BYTE IS 0
00006 4007 CE F424 WAIT1  LDX      #62500   THIS LØØP GENERATES
00007 400A 09      WAIT2  DEX               A DELAY OF ARG*1 SEC
00008 400B 26 FD          BNE      WAIT2
00009 400D 5A             DEC B             DECREMENT THE ARG
00010 400E 26 F7          BNE      WAIT1
00011 4010 4F             CLR A
00012 4011 FE 0117        LDX ...
00013                     *RETURN THE VALUE
00014 4014 6E 00          JMP      X
00015                     END
TØTAL ERRØRS 00000
```

NOTE: This function was developed using the Altair 680 Assembly Language Development System.

The following BASIC program rings the Teletype bell at 10 second intervals by calling the USR function above.

```
 5 REM   SET UP USR ADDRESS
1Ø POKE 287,64
2Ø POKE 288,0
25 REM RING THE BELL
3Ø PRINT CHR$(7);
35 REM DELAY 10 SECONDS
4Ø X = USR(10)
45 REM DO IT AGAIN
5Ø GOTO 3Ø
```

680 BASIC I/O Patch Points

Altair 680 BASIC calls routines in the 680 PROM Monitor to perform I/O transfers. The following routines are necessary and the address of their calls are given.

1) INCH - Input character routine. Reads character from terminal, strips parity, and returns the resultant 7 bit ASCII character in accumulator B. Called from location Ø41F (hex).
2) OUTCH - Output character routine. Sends the ASCII character in accumulator B to the terminal. Called from location Ø2AD (hex).
3) POLCAT - Poll for character routine. Checks input status of terminal. Returns carry set if character has been typed. Returns carry clear if no character has been typed. Called from location Ø618 (hex).

Baudot Control - C

The Baudot version of the PROM Monitor supports only those Baudot Teletypes wired for half-duplex operation. It is therefore impossible to type a control - C while BASIC is doing output. Consequently, BASIC checks the Baudot bit at location FØØ2 and if it indicates the presence of a Baudot terminal, any character typed while BASIC is executing a program will be interpreted as a control - C.

MITS ALTAIR BASIC

REFERENCE MANUAL

Table of Contents:

MITS
"Creative Electronics"
P.O. BOX 8636
ALBUQUERQUE, NEW MEXICO 87108

ALTAIR — MITS — BASIC

Supplement & Errata

The following are additions and corrections to the ALTAIR BASIC REFERENCE MANUAL. Be sure to read this over carefully before continuing.

1) If you are loading BASIC from paper tape, be sure your Serial I/O board is strapped for eight data bits and no parity bit.

2) On page 53 in Appendix C, the meaning for an "OS" error should read:

   Out of String Space. Allocate more string space by using the "CLEAR" command with an argument (see page 42), and then run your program again. If you cannot allocate more string space, try using smaller strings or less string variables.

3) On page 42, under the "CLEAR" command, It is stated that "CLEAR" with no argument sets the amount of string space to 200 bytes. This is incorrect. "CLEAR" with no argument leaves the amount of string space unchanged. When BASIC is brought up, the amount of string space is initially set to 50 bytes.

4) On page 30, under the "DATA" statement, the sentence "IN THE 4K VERSION OF BASIC, DATA STATEMENTS MUST BE THE FIRST STATEMENTS ON A LINE," should be changed to read, "IN THE 4K VERSION OF BASIC, A DATA STATEMENT MUST BE ALONE ON A LINE."

5) If you desire to use a terminal interfaced to the ALTAIR with a Parallel I/O board as your system console, you should load from the ACR interface (wired for address 6). Use the ACR load procedure described in Appendix A, except that you should raise switches 15 & 13 when you start the boot. The Parallel I/O board must be strapped to address 0.

6) If you get a checksum error while loading BASIC from a paper tape or a cassette, you may be able to restart the boot loader at location 0 with the appropriate sense switch settings. This depends on when the error occurs. The boot loader is not written over until the last block of BASIC is being read; which occurs during approximately the last two feet of a paper tape, or the last 10 to 15 seconds of a cassette. If the checksum error occurs during the reading of the last block of BASIC, the boot will be overwritten and you will have to key it in again.

7) The number of nulls punched after a carriage return/line feed does not need to be set >=3 for Teletypes or >=6 for 30 CPS paper tape terminals, as described under the "NULL" command on page 23 of the BASIC manual. In almost all cases, no extra nulls need be punched after a CR/LF on Teletypes, and a setting of nulls to 3 should be sufficient for 30 CPS paper tape terminals. If any problems occur when reading tape (the first few characters of lines are lost), change the null setting to 1 for Teletypes and 4 for 30 CPS terminals.

8) If you have any problems loading BASIC, check to make sure that your terminal interface board (SIO or PIO) is working properly. Key in the appropriate echo program from below, and start it at location zero. Each character typed should be typed or displayed on your terminal. If this is not the case, first be sure that you are using the correct echo program. If you are using the correct program, but it is not functioning properly, then most likely the interface board or the terminal is not operating correctly.

In the following program listings, the number to the left of the slash is the octal address and the number to the right is the octal code for that address.

FOR REV 0 SERIAL I/O BOARDS WITHOUT THE STATUS BIT MODIFICATION

0 / 333	1 / 000	2 / 346
3 / 040	4 / 312	5 / 000
6 / 000	7 / 333	10 / 001
11 / 323	12 / 001	13 / 303
14 / 000	15 / 000	

FOR REV 1 SERIAL I/O BOARDS (AND REV 0 MODIFIED BOARDS)

0 / 333	1 / 000	2 / 017
3 / 332	4 / 000	5 / 000
6 / 333	7 / 001	10 / 323
11 / 001	12 / 303	13 / 000
14 / 000		

FOR PARALLEL I/O BOARDS

0 / 333	1 / 000	2 / 346
3 / 002	4 / 312	5 / 000
6 / 000	7 / 333	10 / 001
11 / 323	12 / 001	13 / 303
14 / 000	15 / 000	

For those of you with the book, MY COMPUTER LIKES ME when i speak in BASIC, by Bob Albrecht, the following information may be helpful.

1) ALTAIR BASIC uses "NEW" instead of "SCR" to delete the current program.

2) Use Control-C to stop execution of a program. Use a carriage-return to stop a program at an "INPUT" statement.

3) You don't need an "END" statement at the end of a BASIC program.

8/25/75

Introduction

Before a computer can perform any useful function, it must be "told" what to do. Unfortunately, at this time, computers are not capable of understanding English or any other "human" language. This is primarily because our languages are rich with ambiguities and implied meanings. The computer must be told precise instructions and the exact sequence of operations to be performed in order to accomplish any specific task. Therefore, in order to facilitate human communication with a computer, programming languages have been developed.

ALTAIR BASIC* is a programming language both easily understood and simple to use. It serves as an excellent "tool" for applications in areas such as business, science and education. With only a few hours of using BASIC, you will find that you can already write programs with an ease that few other computer languages can duplicate.

Originally developed at Dartmouth University, BASIC language has found wide acceptance in the computer field. Although it is one of the simplest computer languages to use, it is very powerful. BASIC uses a small set of common English words as its "commands". Designed specifically as an "interactive" language, you can give a command such as "PRINT 2 + 2", and ALTAIR BASIC will immediately reply with "4". It isn't necessary to submit a card deck with your program on it and then wait hours for the results. Instead the full power of the ALTAIR is "at your fingertips".

Generally, if the computer does not solve a particular problem the way you expected it to, there is a "Bug" or error in your program, or else there is an error in the data which the program used to calculate its answer. If you encounter any errors in BASIC itself, please let us know and we'll see that it's corrected. Write a letter to us containing the following information:

1) System Configuration

2) Version of BASIC

3) A detailed description of the error
Include all pertinent information such as a listing of the program in which the error occurred, the data placed into the program and BASIC's printout.

All of the information listed above will be necessary in order to properly evaluate the problem and correct it as quickly as possible. We wish to maintain as high a level of quality as possible with all of our ALTAIR software.

* *BASIC is a registered trademark of Dartmouth University.*

We hope that you enjoy ALTAIR BASIC, and are successful in using it to solve all of your programming needs.

In order to maintain a maximum quality level in our documentation, we will be continuously revising this manual. If you have any suggestions on how we can improve it, please let us know.

If you are already familiar with BASIC programming, the following section may be skipped. Turn directly to the Reference Material on page 22.

If any immediate problems with MITS software are encountered, feel free to give us a call at (505) 265-7553. The Software Department is at Ext. 3; and the joint authors of the ALTAIR BASIC Interpreter, Bill Gates, Paul Allen and Monte Davidoff, will be glad to assist you.

GETTING STARTED WITH BASIC

MITS
"Creative Electronics"

This section is not intended to be a detailed course in BASIC programming. It will, however, serve as an excellent introduction for those of you unfamiliar with the language.

The text here will introduce the primary concepts and uses of BASIC enough to get you started writing programs. For further reading suggestions, see Appendix M.

If your ALTAIR does not have BASIC loaded and running, follow the procedures in Appendices A & B to bring it up.

We recommend that you try each example in this section as it is presented. This will enhance your "feel" for BASIC and how it is used.

Once your I/O device has typed " OK ", you are ready to use ALTAIR BASIC.

NOTE: All commands to ALTAIR BASIC should end with a carriage return. The carriage return tells BASIC that you have finished typing the command. If you make a typing error, type a back-arrow (←), usually shift/O, or an underline to eliminate the last character. Repeated use of " ← " will eliminate previous characters. An at-sign (@) will eliminate the entire line that you are typing.

Now, try typing in the following:

```
PRINT 10-4   (end with carriage return)
```

ALTAIR BASIC will immediately print:

```
6

OK
```

The print statement you typed in was executed as soon as you hit the carriage return key. BASIC evaluated the formula after the "PRINT" and then typed out its value, in this case 6.

Now try typing in this:

```
PRINT 1/2,3*10          ("*" means multiply, "/" means divide)
```

ALTAIR BASIC will print:

```
.5            30
```

As you can see, ALTAIR BASIC can do division and multiplication as well as subtraction. Note how a " , " (comma) was used in the print command to print two values instead of just one. The comma divides the 72 character line into 5 columns, each 14 characters wide. The last two of the positions on the line are not used.. The result is a " , " causes BASIC to skip to the next 14 column field on the terminal, where the value 30 was printed.

Commands such as the "PRINT" statements you have just typed in are called Direct Commands. There is another type of command called an Indirect Command. Every Indirect command begins with a Line Number. A Line Number is any integer from 0 to 65529.

Try typing in the following lines:

```
10 PRINT 2+3
20 PRINT 2-3
```

A sequence of Indirect Commands is called a "Program". Instead of executing indirect statements immediately, ALTAIR BASIC saves Indirect Commands in the ALTAIR's memory. When you type in RUN , BASIC will execute the lowest numbered indirect statement that has been typed in first, then the next highest, etc. for as many as were typed in.

Suppose we type in RUN now:

```
RUN
```

ALTAIR BASIC will type out:

```
 5
-1

OK
```

In the example above, we typed in line 10 first and line 20 second. However, it makes no difference in what order you type in indirect statements. BASIC always puts them into correct numerical order according to the Line Number.

If we want a listing of the complete program currently in memory, we type in LIST . Type this in:

```
LIST
```

ALTAIR BASIC will reply with:

```
10 PRINT 2+3
20 PRINT 2-3
OK
```

Sometimes it is desirable to delete a line of a program altogether. This is accomplished by typing the Line Number of the line we wish to delete, followed only by a carriage return.

Type in the following:

```
10
LIST
```

ALTAIR BASIC will reply with:

```
20 PRINT 2-3
OK
```

We have now deleted line 10 from the program. There is no way to get it back. To insert a new line 10, just type in 10 followed by the statement we want BASIC to execute.

Type in the following:

```
10 PRINT 2*3
LIST
```

ALTAIR BASIC will reply with:

```
10 PRINT 2*3
20 PRINT 2-3
OK
```

There is an easier way to replace line 10 than deleting it and then inserting a new line. You can do this by just typing the new line 10 and hitting the carriage return. BASIC throws away the old line 10 and replaces it with the new one.

Type in the following:

```
10 PRINT 3-3
LIST
```

ALTAIR BASIC will reply with:

```
10 PRINT 3-3
20 PRINT 2-3
OK
```

It is not recommended that lines be numbered consecutively. It may become necessary to insert a new line between two existing lines. An increment of 10 between line numbers is generally sufficient.

If you want to erase the complete program currently stored in memory, type in " NEW ". If you are finished running one program and are about to read in a new one, be sure to type in " NEW " first. This should be done in order to prevent a mixture of the old and new programs.

Type in the following:

```
NEW
```

ALTAIR BASIC will reply with:

```
OK
```


Now type in:

```
LIST
```

ALTAIR BASIC will reply with:

```
OK
```

Often it is desirable to include text along with answers that are printed out, in order to explain the meaning of the numbers.

Type in the following:

```
PRINT "ONE THIRD IS EQUAL TO",1/3
```

ALTAIR BASIC will reply with:

```
ONE THIRD IS EQUAL TO         .333333

OK
```

As explained earlier, including a " , " in a print statement causes it to space over to the next fourteen column field before the value following the " , " is printed.

If we use a " ; " instead of a comma, the value next will be printed immediately following the previous value.

NOTE: Numbers are always printed with at least one trailing space. Any text to be printed is always to be enclosed in double quotes.

Try the following examples;

A)
```
PRINT "ONE THIRD IS EQUAL TO";1/3
ONE THIRD IS EQUAL TO .333333

OK
```

B)
```
PRINT 1,2,3
 1             2             3

OK
```

C)
```
PRINT 1;2;3
 1  2  3

OK
```

D)
```
PRINT -1;2;-3
-1  2 -3
```

OK

We will digress for a moment to explain the format of numbers in ALTAIR BASIC. Numbers are stored internally to over six digits of accuracy. When a number is printed, only six digits are shown. Every number may also have an exponent (a power of ten scaling factor).

The largest number that may be represented in ALTAIR BASIC is $1.70141*10^{38}$, while the smallest positive number is $2.93874*10^{-39}$.

When a number is printed, the following rules are used to determine the exact format:

1) If the number is negative, a minus sign (-) is printed. If the number is positive, a space is printed.

2) If the absolute value of the number is an integer in the range 0 to 999999, it is printed as an integer.

3) If the absolute value of the number is greater than or equal to .1 and less than or equal to 999999, it is printed in fixed point notation, with no exponent.

4) If the number does not fall under categories 2 or 3, scientific notation is used.

Scientific notation is formatted as follows: SX.XXXXXESTT . (each X being some integer 0 to 9)

The leading "S" is the sign of the number, a space for a positive number and a " - " for a negative one. One non-zero digit is printed before the decimal point. This is followed by the decimal point and then the other five digits of the mantissa. An "E" is then printed (for exponent), followed by the sign (S) of the exponent; then the two digits (TT) of the exponent itself. Leading zeroes are never printed; i.e. the digit before the decimal is never zero. Also, trailing zeroes are never printed. If there is only one digit to print after all trailing zeroes are suppressed, no decimal point is printed. The exponent sign will be " + " for positive and " - " for negative. Two digits of the exponent are always printed; that is zeroes are not suppressed in the exponent field. The value of any number expressed thus is the number to the left of the "E" times 10 raised to the power of the number to the right of the "E".

No matter what format is used, a space is always printed following a number. The 8K version of BASIC checks to see if the entire number will fit on the current line. If not, a carriage return/line feed is executed before printing the number.

The following are examples of various numbers and the output format ALTAIR BASIC will place them into:

NUMBER	OUTPUT FORMAT
+1	1
-1	-1
6523	6523
-23.460	-23.46
1E20	1E+20
-12.3456E-7	-1.23456E-06
1.234567E-10	1.23457E-10
1000000	1E+06
999999	999999
.1	.1
.01	1E-02
.000123	1.23E-04

A number input from the terminal or a numeric constant used in a BASIC program may have as many digits as desired, up to the maximum length of a line (72 characters). However, only the first 7 digits are significant, and the seventh digit is rounded up.

```
PRINT 1.23456789012345678901
 1.23457

OK
```

The following is an example of a program that reads a value from the terminal and uses that value to calculate and print a result:

```
10 INPUT R
20 PRINT 3.14159*R*R
RUN
? 10
 314.159

OK
```

Here's what's happening. When BASIC encounters the input statement, it types a question mark (?) on the terminal and then waits for you to type in a number. When you do (in the above example 10 was typed), execution continues with the next statement in the program after the variable (R) has been set (in this case to 10). In the above example, line 20 would now be executed. When the formula after the PRINT statement is evaluated, the value 10 is substituted for the variable R each time R appears in the formula. Therefore, the formula becomes 3.14159*10*10, or 314.159.

If you haven't already guessed, what the program above actually does is to calculate the area of a circle with the radius "R".

If we wanted to calculate the area of various circles, we could keep re-running the program over each time for each successive circle. But, there's an easier way to do it simply by adding another line to the program as follows:

```
30 GOTO 10
RUN
? 10
 314.159
? 3
 28.2743
? 4.7
 69.3977
?

OK
```

By putting a " GOTO " statement on the end of our program, we have caused it to go back to line 10 after it prints each answer for the successive circles. This could have gone on indefinitely, but we decided to stop after calculating the area for three circles. This was accomplished by typing a carriage return to the input statement (thus a blank line).

NOTE: Typing a carriage return to an input statement in the 4K version of BASIC will cause a SN error (see Reference Material).

The letter "R" in the program we just used was termed a "variable". A variable name can be any alphabetic character and may be followed by any alphanumeric character.
In the 4K version of BASIC, the second character must be numeric or omitted. In the 8K version of BASIC, any alphanumeric characters after the first two are ignored. An alphanumeric character is any letter (A-Z) or any number (0-9).

Below are some examples of legal and illegal variable names;

LEGAL	ILLEGAL
IN 4K VERSION	
A	% (1st character must be alphabetic)
Z1	Z1A (variable name too long)
	QR (2nd character must be numeric)
IN 8K VERSION	
TP	TO (variable names cannot be reserved words)
PSTG$	RGOTO (variable names cannot contain reserved words)
COUNT	

The words used as BASIC statements are "reserved" for this specific purpose. You cannot use these words as variable names or inside of any variable name. For instance, "FEND" would be illegal because "END" is a reserved word.

The following is a list of the reserved words in ALTAIR BASIC:

4K RESERVED WORDS

ABS CLEAR DATA DIM END FOR GOSUB GOTO IF INPUT

INT LET LIST NEW NEXT PRINT READ REM RESTORE

RETURN RND RUN SGN SIN SQR STEP STOP TAB(THEN

TO USR

8K RESERVED WORDS INCLUDE ALL THOSE ABOVE, AND IN ADDITION

ASC AND ATN CHR$ CLOAD CONT COS CSAVE DEF EXP

FN FRE INP LEFT$ LEN LOG MID$ NULL ON OR NOT

OUT PEEK POKE POS RIGHT$ SPC(STR$ TAN VAL WAIT

Remember, in the 4K version of BASIC variable names are only a letter or a letter followed by a number. Therefore, there is no possibility of a conflict with a reserved word.

Besides having values assigned to variables with an input statement, you can also set the value of a variable with a LET or assignment statement.

Try the following examples;

```
A=5

OK

PRINT A,A*2
 5             10

OK
LET Z=7

OK

PRINT Z, Z-A
 7             2

OK
```

As can be seen from the examples, the "LET" is optional in an assignment statement.

BASIC "remembers" the values that have been assigned to variables using this type of statement. This "remembering" process uses space in the ALTAIR's memory to store the data.

The values of variables are thrown away and the space in memory used to store them is released when one of four things occur:

1) A new line is typed into the program or an old line is deleted

2) A CLEAR command is typed in

3) A RUN command is typed in

4) NEW is typed in

Another important fact is that if a variable is encountered in a formula before it is assigned a value, it is automatically assigned the value zero. Zero is then substituted as the value of the variable in the particular formula. Try the example below:

```
PRINT Q,Q+2,Q*2
 0          2          0

OK
```

Another statement is the REM statement. REM is short for remark. This statement is used to insert comments or notes into a program. When BASIC encounters a REM statement the rest of the line is ignored.

This serves mainly as an aid for the programmer himself, and serves no useful function as far as the operation of the program in solving a particular problem.

Suppose we wanted to write a program to check if a number is zero or not. With the statements we've gone over so far this could not be done. What is needed is a statement which can be used to conditionally branch to another statement. The "IF-THEN" statement does just that.

Try typing in the following program: (remember, type NEW first)

```
10 INPUT B
20 IF B=0 THEN 50
30 PRINT "NON-ZERO"
40 GOTO 10
50 PRINT "ZERO"
60 GOTO 10
```

When this program is typed into the ALTAIR and run, it will ask for a value for B. Type any value you wish in. The ALTAIR will then come to the "IF" statement. Between the "IF" and the "THEN" portion of the statement there are two expressions separated by a relation.

A relation is one of the following six symbols:

RELATION	MEANING
=	EQUAL TO
>	GREATER THAN
<	LESS THAN
<>	NOT EQUAL TO
<=	LESS THAN OR EQUAL TO
=>	GREATER THAN OR EQUAL TO

The IF statement is either true or false, depending upon whether the two expressions satisfy the relation or not. For example, in the program we just did, if 0 was typed in for B the IF statement would be true because 0=0. In this case, since the number after the THEN is 50, execution of the program would continue at line 50. Therefore, "ZERO" would be printed and then the program would jump back to line 10 (because of the GOTO statement in line 60).

Suppose a 1 was typed in for B. Since 1=0 is false, the IF statement would be false and the program would continue execution with the next line. Therefore, "NON-ZERO" would be printed and the GOTO in line 40 would send the program back to line 10.

Now try the following program for comparing two numbers:

```
10 INPUT A,B
20 IF A<=B THEN 50
30 PRINT "A IS BIGGER"
40 GOTO 10
50 IF A<B THEN 80
60 PRINT "THEY ARE THE SAME"
70 GOTO 10
80 PRINT "B IS BIGGER"
90 GOTO 10
```

When this program is run, line 10 will input two numbers from the terminal. At line 20, if A is greater than B, A<=B will be false. This will cause the next statement to be executed, printing "A IS BIGGER" and then line 40 sends the computer back to line 10 to begin again.

At line 20, if A has the same value as B, A<=B is true so we go to line 50. At line 50, since A has the same value as B, A<B is false; therefore, we go to the following statement and print "THEY ARE THE SAME". Then line 70 sends us back to the beginning again.

At line 20, if A is smaller than B, A<=B is true so we go to line 50. At line 50, A<B will be true so we then go to line 80. "B IS BIGGER" is then printed and again we go back to the beginning.

Try running the last two programs several times. It may make it easier to understand if you try writing your own program at this time using the IF-THEN statement. Actually trying programs of your own is the quickest and easiest way to understand how BASIC works. Remember, to stop these programs just give a carriage return to the input statement.

One advantage of computers is their ability to perform repetitive tasks. Let's take a closer look and see how this works.

Suppose we want a table of square roots from 1 to 10. The BASIC function for square root is "SQR"; the form being SQR(X), X being the number you wish the square root calculated from. We could write the program as follows:

```
10 PRINT 1,SQR(1)
20 PRINT 2,SQR(2)
30 PRINT 3,SQR(3)
40 PRINT 4,SQR(4)
50 PRINT 5,SQR(5)
60 PRINT 6,SQR(6)
70 PRINT 7,SQR(7)
80 PRINT 8,SQR(8)
90 PRINT 9,SQR(9)
100 PRINT 10,SQR(10)
```

This program will do the job; however, it is terribly inefficient. We can improve the program tremendously by using the IF statement just introduced as follows:

```
10 N=1
20 PRINT N,SQR(N)
30 N=N+1
40 IF N<=10 THEN 20
```

When this program is run, its output will look exactly like that of the 10 statement program above it. Let's look at how it works.

At line 10 we have a LET statement which sets the value of the variable N at 1. At line 20 we print N and the square root of N using its current value. It thus becomes 20 PRINT 1,SQR(1), and this calculation is printed out.

At line 30 we use what will appear at first to be a rather unusual LET statement. Mathematically, the statement N=N+1 is nonsense. However, the important thing to remember is that in a LET statement, the symbol " = " does not signify equality. In this case " = " means "to be replaced with". All the statement does is to take the current value of N and add 1 to it. Thus, after the first time through line 30, N becomes 2.

At line 40, since N now equals 2, N<=10 is true so the THEN portion branches us back to line 20, with N now at a value of 2.

The overall result is that lines 20 through 40 are repeated, each time adding 1 to the value of N. When N finally equals 10 at line 20, the next line will increment it to 11. This results in a false statement at line 40, and since there are no further statements to the program it stops.

This technique is referred to as "looping" or "iteration". Since it is used quite extensively in programming, there are special BASIC statements for using it. We can show these with the following program.


```
10 FOR N=1 TO 10
20 PRINT N,SQR(N)
30 NEXT N
```

The output of the program listed above will be exactly the same as the previous two programs.

At line 10, N is set to equal 1. Line 20 causes the value of N and the square root of N to be printed. At line 30 we see a new type of statement. The "NEXT N" statement causes one to be added to N, and then if N<=10 we go back to the statement following the "FOR" statement. The overall operation then is the same as with the previous program.

Notice that the variable following the "FOR" is exactly the same as the variable after the "NEXT". There is nothing special about the N in this case. Any variable could be used, as long as they are the same in both the "FOR" and the "NEXT" statements. For instance, "Z1" could be substituted everywhere there is an "N" in the above program and it would function exactly the same.

Suppose we wanted to print a table of square roots from 10 to 20, only counting by two's. The following program would perform this task:

```
10 N=10
20 PRINT N,SQR(N)
30 N=N+2
40 IF N<=20 THEN 20
```

Note the similar structure between this program and the one listed on page 12 for printing square roots for the numbers 1 to 10. This program can also be written using the "FOR" loop just introduced.

```
10 FOR N=10 TO 20 STEP 2
20 PRINT N,SQR(N)
30 NEXT N
```

Notice that the only major difference between this program and the previous one using "FOR" loops is the addition of the "STEP 2" clause.

This tells BASIC to add 2 to N each time, instead of 1 as in the previous program. If no "STEP" is given in a "FOR" statement, BASIC assumes that one is to be added each time. The "STEP" can be followed by any expression.

Suppose we wanted to count backwards from 10 to 1. A program for doing this would be as follows:

```
10 I=10
20 PRINT I
30 I=I-1
40 IF I>=1 THEN 20
```

Notice that we are now checking to see that I is greater than or equal to the final value. The reason is that we are now counting by a negative number. In the previous examples it was the opposite, so we were checking for a variable less than or equal to the final value.

```
110 IF N = INT(N) THEN 140
120 PRINT "SORRY, NUMBER MUST BE AN INTEGER.  TRY AGAIN."
130 GOTO 100
140 RETURN
```

What this program does is to ask for two numbers which must be integers, and then prints the sum of the two. The subroutine in this program is lines 100 to 130. The subroutine asks for a number, and if it is not an integer, asks for a number again. It will continue to ask until an integer value is typed in.

The main program prints " WHAT IS THE NUMBER ", and then calls the subroutine to get the value of the number into N. When the subroutine returns (to line 40), the value input is saved in the variable T. This is done so that when the subroutine is called a second time, the value of the first number will not be lost.

" WHAT IS THE SECOND NUMBER " is then printed, and the second value is entered when the subroutine is again called.

When the subroutine returns the second time, " THE SUM OF THE TWO NUMBERS IS " is printed, followed by the value of their sum. T contains the value of the first number that was entered and N contains the value of the second number.

The next statement in the program is a "STOP" statement. This causes the program to stop execution at line 90. If the "STOP" statement was not included in the program, we would "fall into" the subroutine at line 100. This is undesirable because we would be asked to input another number. If we did, the subroutine would try to return; and since there was no "GOSUB" which called the subroutine, an RG error would occur. Each "GOSUB" executed in a program should have a matching "RETURN" executed later, and the opposite applies, i.e. a "RETURN" should be encountered only if it is part of a subroutine which has been called by a "GOSUB".

Either "STOP" or "END" can be used to separate a program from its subroutines. In the 4K version of BASIC, there is no difference between the "STOP" and the "END". In the 8K version, "STOP" will print a message saying at what line the "STOP" was encountered.

Suppose you had to enter numbers to your program that didn't change each time the program was run, but you would like it to be easy to change them if necessary. BASIC contains special statements for this purpose, called the "READ" and "DATA" statements.

Consider the following program:

```
10 PRINT "GUESS A NUMBER";
20 INPUT G
30 READ D
40 IF D=-999999 THEN 90
50 IF D<>G THEN 30
60 PRINT "YOU ARE CORRECT"
70 END
90 PRINT "BAD GUESS, TRY AGAIN."
95 RESTORE
```



```
100 GOTO 10
110 DATA 1,393,-39,28,391,-8,0,3.14,90
120 DATA 89,5,10,15,-34,-999999
```

This is what happens when this program is run. When the "READ" statement is encountered, the effect is the same as an INPUT statement. But, instead of getting a number from the terminal, a number is read from the "DATA" statements.

The first time a number is needed for a READ, the first number in the first DATA statement is returned. The second time one is needed, the second number in the first DATA statement is returned. When the entire contents of the first DATA statement have been read in this manner, the second DATA statement will then be used. DATA is always read sequentially in this manner, and there may be any number of DATA statements in your program.

The purpose of this program is to play a little game in which you try to guess one of the numbers contained in the DATA statements. For each guess that is typed in, we read through all of the numbers in the DATA statements until we find one that matches the guess.

If more values are read than there are numbers in the DATA statements, an out of data (OD) error occurs. That is why in line 40 we check to see if -999999 was read. This is not one of the numbers to be matched, but is used as a flag to indicate that all of the data (possible correct guesses) has been read. Therefore, if -999999 was read, we know that the guess given was incorrect.

Before going back to line 10 for another guess, we need to make the READ's begin with the first piece of data again. This is the function of the "RESTORE". After the RESTORE is encountered, the next piece of data read will be the first piece in the first DATA statement again.

DATA statements may be placed anywhere within the program. Only READ statements make use of the DATA statements in a program, and any other time they are encountered during program execution they will be ignored.

THE FOLLOWING INFORMATION APPLIES TO THE 8K VERSION OF BASIC ONLY

A list of characters is referred to as a "String". MITS, ALTAIR, and THIS IS A TEST are all strings. Like numeric variables, string variables can be assigned specific values. String variables are distinguished from numeric variables by a "$" after the variable name.

For example, try the following:

```
A$="ALTAIR 8800"

OK
PRINT A$
ALTAIR 8800

OK
```

In this example, we set the string variable A$ to the string value "ALTAIR 8800". Note that we also enclosed the character string to be assigned to A$ in quotes.

Now that we have set A$ to a string value, we can find out what the length of this value is (the number of characters it contains). We do this as follows:

```
PRINT LEN(A$),LEN("MITS")
 11            4

OK
```

The "LEN" function returns an integer equal to the number of characters in a string.

The number of characters in a string expression may range from 0 to 255. A string which contains 0 characters is called the "NULL" string. Before a string variable is set to a value in the program, it is initialized to the null string. Printing a null string on the terminal will cause no characters to be printed, and the print head or cursor will not be advanced to the next column. Try the following:

```
PRINT LEN(Q$);Q$;3
 0  3

OK
```

Another way to create the null string is: Q$=""

Setting a string variable to the null string can be used to free up the string space used by a non-null string variable.

Often it is desirable to access parts of a string and manipulate them. Now that we have set A$ to "ALTAIR 8800", we might want to print out only the first six characters of A$. We would do so like this:

```
PRINT LEFT$(A$,6)
ALTAIR

OK
```

"LEFT$" is a string function which returns a string composed of the leftmost N characters of its string argument. Here's another example:

```
FOR N=1 TO LEN(A$):PRINT LEFT$(A$,N):NEXT N
A
AL
ALT
ALTA
ALTAI
ALTAIR
ALTAIR
ALTAIR 8
ALTAIR 88
```



```
ALTAIR 880
ALTAIR 8800

OK
```

Since A$ has 11 characters, this loop will be executed with N=1,2,3,...,10,11. The first time through only the first chatacter will be printed, the second time the first two characters will be printed, etc.

There is another string function called "RIGHT$" which returns the right N characters from a string expression. Try substituting "RIGHT$" for "LEFT$" in the previous example and see what happens.

There is also a string function which allows us to take characters from the middle of a string. Try the following:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N):NEXT N
ALTAIR 8800
LTAIR 8800
TAIR 8800
AIR 8800
IR 8800
R 8800
 8800
8800
800
00
0

OK
```

"MID$" returns a string starting at the Nth position of A$ to the end (last character) of A$. The first position of the string is position 1 and the last possible position of a string is position 255.

Very often it is desirable to extract only the Nth character from a string. This can be done by calling MID$ with three arguments. The third argument specifies the number of characters to return.

For example:

```
FOR N=1 TO LEN(A$):PRINT MID$(A$,N,1),MID$(A$,N,2):NEXT N
A              AL
L              LT
T              TA
A              AI
I              IR
R              R
                8
8              88
8              80
0              00
0              0

OK
```

See the Reference Material for more details on the workings of "LEFT$", "RIGHT$" and "MID$".

Strings may also be concatenated (put or joined together) through the use of the "+" operator. Try the following:

```
B$="MITS"+" "+A$

OK
PRINT B$
MITS ALTAIR 8800

OK
```

Concatenation is especially useful if you wish to take a string apart and then put it back together with slight modifications. For instance:

```
C$=LEFT$(B$,4)+"-"+MID$(B$,6,6)+"-"+RIGHT$(B$,4)

OK
PRINT C$
MITS-ALTAIR-8800

OK
```

Sometimes it is desirable to convert a number to its string representation and vice-versa. "VAL" and "STR$" perform these functions. Try the following:

```
STRING$="567.8"

OK
PRINT VAL(STRING$)
 567.8

OK
STRING$=STR$(3.1415)

OK
PRINT STRING$,LEFT$(STRING$,5)
 3.1415        3.14

OK
```

"STR$" can be used to perform formatted I/O on numbers. You can convert a number to a string and then use LEFT$, RIGHT$, MID$ and concatenation to reformat the number as desired.

"STR$" can also be used to conveniently find out how many print columns a number will take. For example:

```
PRINT LEN(STR$(3.157))
 6
```



```
OK
```

If you have an application where a user is typing in a question such as "WHAT IS THE VOLUME OF A CYLINDER OF RADIUS 5.36 FEET, OF HEIGHT 5.1 FEET?" you can use "VAL" to extract the numeric values 5.36 and 5.1 from the question. For further functions "CHR$" and "ASC" see Appendix K.

The following program sorts a list of string data and prints out the sorted list. This program is very similar to the one given earlier for sorting a numeric list.

```
100 DIM A$(15):REM ALLOCATE SPACE FOR STRING MATRIX
110 FOR I=1 TO 15:READ A$(I):NEXT I:REM READ IN STRINGS
120 F=0:I=1:REM SET EXCHANGE FLAG TO ZERO AND SUBSCRIPT TO 1
130 IF A$(I)<=A$(I+1) THEN 180:REM DON'T EXCHANGE IF ELEMENTS
    IN ORDER
140 T$=A$(I+1):REM USE T$ TO SAVE A$(I+1)
150 A$(I+1)=A$(I):REM EXCHANGE TWO CONSECUTIVE ELEMENTS
160 A$(I)=T$
170 F=1:REM FLAG THAT WE EXCHANGED TWO ELEMENTS
180 I=I+1: IF I<15 GOTO 130
185 REM ONCE WE HAVE MADE A PASS THRU ALL ELEMENTS, CHECK
187 REM TO SEE IF WE EXCHANGED ANY.  IF NOT, DONE SORTING.
190 IF F THEN 120:REM EQUIVALENT TO IF F<>0 THEN 120
200 FOR I=1 TO 15:PRINT A$(I):NEXT I: REM PRINT SORTED LIST
210 REM STRING DATA FOLLOWS
220 DATA APPLE,DOG,CAT,MITS,ALTAIR,RANDOM
230 DATA MONDAY,"***ANSWER***"," FOO"
240 DATA COMPUTER,   FOO,ELP,MILWAUKEE,SEATTLE,ALBUQUERQUE
```

COMMANDS

A command is usually given after BASIC has typed OK. This is called the "Command Level". Commands may be used as program statements. Certain commands, such as LIST, NEW and CLOAD will terminate program execution when they finish.

NAME	EXAMPLE	PURPOSE/USE
CLEAR	*(SEE PAGE 42 FOR EXAMPLES AND EXPLANATION)	
LIST	LIST LIST 100	Lists current program optionally starting at specified line. List can be control-C'd (BASIC will finish listing the current line)
NULL	NULL 3	(Null command only in 8K version, but paragraph applicable to 4K version also) Sets the number of null (ASCII 0) characters printed after a carriage return/line feed. The number of nulls printed may be set from 0 to 71. This is a must for hardcopy terminals that require a delay after a CRLF.* It is necessary to set the number of nulls typed on CRLF to 0 before a paper tape of a program is read in from a Teletype *(TELETYPE is a registered trademark of the TELETYPE CORPORATION)*. In the 8K version, use the null command to set the number of nulls to zero. In the 4K version, this is accomplished by patching location 46 octal to contain the number of nulls to be typed plus 1. (Depositing a 1 in location 46 would set the number of nulls typed to zero.) When you punch a paper tape of a program using the list command, null should be set >=3 for 10 CPS terminals, >=6 for 30 CPS terminals. When not making a tape, we recommend that you use a null setting of 0 or 1 for Teletypes, and 2 or 3 for hard copy 30 CPS terminals. A setting of 0 will work with Teletype compatible CRT's.
RUN	RUN	Starts execution of the program currently in memory at the lowest numbered statement. Run deletes all variables (does a CLEAR) and restores DATA. If you have stopped your program and wish to continue execution at some point in the program, use a direct GOTO statement to start execution of your program at the desired line.

*CRLF=carriage return/line feed

	RUN 200	(8K version only) optionally starting at the specified line number
NEW	NEW	Deletes current program and all variables

THE FOLLOWING COMMANDS ARE IN THE 8K VERSION ONLY

CONT	CONT	Continues program execution after a control/C is typed or a STOP statement is executed. You cannot continue after any error, after modifying your program, or before your program has been run. One of the main purposes of CONT is debugging. Suppose at some point after running your program, nothing is printed. This may be because your program is performing some time consuming calculation, but it may be because you have fallen into an "infinite loop". An infinite loop is a series of BASIC statements from which there is no escape. The ALTAIR will keep executing the series of statements over and over, until you intervene or until power to the ALTAIR is cut off. If you suspect your program is in an infinite loop, type in a control/C. In the 8K version, the line number of the statement BASIC was executing will be typed out. After BASIC has typed out OK, you can use PRINT to type out some of the values of your variables. After examining these values you may become satisfied that your program is functioning correctly. You should then type in CONT to continue executing your program where it left off, or type a direct GOTO statement to resume execution of the program at a different line. You could also use assignment (LET) statements to set some of your variables to different values. Remember, if you control/C a program and expect to continue it later, you must not get any errors or type in any new program lines. If you do, you won't be able to continue and will get a "CN" (continue not) error. It is impossible to continue a direct command. CONT always resumes execution at the next statement to be executed in your program when control/C was typed.

THE FOLLOWING TWO COMMANDS ARE AVAILABLE IN THE 8K CASSETTE VERSION ONLY

CLOAD	CLOAD P	Loads the program named P from the cassette tape. A NEW command is automatically done before the CLOAD command is executed. When done, the CLOAD will type out OK as usual. The one-character program designator may be any printing character. CSAVE and CLOAD use I/O ports 6 & 7. See Appendix I for more information.
CSAVE	CSAVE P	Saves on cassette tape the current program in the ALTAIR's memory. The program in memory is left unchanged. More than one program may be stored on cassette using this command. CSAVE and CLOAD use I/O ports 6 & 7. See Appendix I for more information

OPERATORS

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	A=100 LET Z=2.5	Assigns a value to a variable The LET is optional
-	B=-A	Negation. Note that 0-A is subtraction, while -A is negation.
↑ *(usually a shift/N)*	130 PRINT X↑3	Exponentiation (8K version) (equal to X*X*X in the sample statement) 0↑0=1 0 to any other power = 0 A↑B, with A negative and B not an integer gives an FC error.
*	140 X=R*(B*D)	Multiplication
/	150 PRINT X/1.3	Division
+	160 Z=R+T+Q	Addition
-	170 J=100-I	Subtraction

RULES FOR EVALUATING EXPRESSIONS:

1) Operations of higher precedence are performed before operations of lower precedence. This means the multiplication and divisions are performed before additions and subtractions. As an example, 2+10/5 equals 4, not 2.4. When operations of equal precedence are found in a formula, the left hand one is executed first: 6-3+5=8, not -2.

2) The order in which operations are performed can always be specified explicitly through the use of parentheses. For instance, to add 5 to 3 and then divide that by 4, we would use (5+3)/4, which equals 2. If instead we had used 5+3/4, we would get 5.75 as a result (5 plus 3/4).

The precedence of operators used in evaluating expressions is as follows, in order beginning with the highest precedence:
(Note: Operators listed on the same line have the same precedence.)

1) FORMULAS ENCLOSED IN PARENTHESIS ARE ALWAYS EVALUATED FIRST

2)	↑	EXPONENTIATION *(8K VERSION ONLY)*
3)	NEGATION	-X WHERE X MAY BE A FORMULA
4)	* /	MULTIPLICATION AND DIVISION
5)	+ -	ADDITION AND SUBTRACTION
6)	RELATIONAL OPERATORS: *(equal precedence for all six)*	= EQUAL <> NOT EQUAL < LESS THAN > GREATER THAN <= LESS THAN OR EQUAL >= GREATER THAN OR EQUAL

(8K VERSION ONLY) (These 3 below are Logical Operators)

7)	NOT	LOGICAL AND BITWISE "NOT" LIKE NEGATION, NOT TAKES ONLY THE FORMULA TO ITS RIGHT AS AN ARGUMENT
8)	AND	LOGICAL AND BITWISE "AND"
9)	OR	LOGICAL AND BITWISE "OR"

In the 4K version of BASIC, relational operators can only be used once in an IF statement. However, in the 8K version a relational expression can be used as part of any expression.

Relational Operator expressions will always have a value of True (-1) or a value of False (0). Therefore, (5=4)=0, (5=5)=-1, (4>5)=0, (4<5)=-1, etc.

The THEN clause of an IF statement is executed whenever the formula after the IF is not equal to 0. That is to say, IF X THEN... is equivalent to IF X<>0 THEN... .

SYMBOL	SAMPLE STATEMENT	PURPOSE/USE
=	10 IF A=15 THEN 40	Expression Equals Expression
<>	70 IF A<>0 THEN 5	Expression Does Not Equal Expression
>	30 IF B>100 THEN 8	Expression Greater Than Expression
<	160 IF B<2 THEN 10	Expression Less Than Expression
<=,=<	180 IF 100<=B+C THEN 10	Expression Less Than Or Equal To Expression
>=,=>	190 IF Q=>R THEN 50	Expression Greater Than Or Equal To Expression
AND	2 IF A<5 AND B<2 THEN 7	*(8K Version only)* If expression 1 (A<5) AND expression 2 (B<2) are both true, then branch to line 7
OR	IF A<1 OR B<2 THEN 2	*(8K Version only)* If either expression 1 (A<1) OR expression 2 (B<2) is true, then branch to line 2
NOT	IF NOT Q3 THEN 4	*(8K Version only)* If expression "NOT Q3" is true (because Q3 is false), then branch to line 4 *Note: NOT -1=0 (NOT true=false)*

AND, OR and NOT can be used for bit manipulation, and for performing boolean operations.

These three operators convert their arguments to sixteen bit, signed two's, complement integers in the range -32768 to +32767. They then perform the specified logical operation on them and return a result within the same range. If the arguments are not in this range, an "FC" error results.

The operations are performed in bitwise fashion, this means that each bit of the result is obtained by examining the bit in the same position for each argument.

The following truth table shows the logical relationship between bits:

OPERATOR	ARG. 1	ARG. 2	RESULT
AND	1	1	1
	0	1	0
	1	0	0
	0	0	0

(cont.)

OPERATOR	ARG. 1	ARG. 2	RESULT
OR	1	1	1
	1	0	1
	0	1	1
	0	0	0
NOT	1	-	0
	0	-	1

EXAMPLES: *(In all of the examples below, leading zeroes on binary numbers are not shown.)*

63 AND 16=16	Since 63 equals binary 111111 and 16 equals binary 10000, the result of the AND is binary 10000 or 16.
15 AND 14=14	15 equals binary 1111 and 14 equals binary 1110, so 15 AND 14 equals binary 1110 or 14.
-1 AND 8=8	-1 equals binary 1111111111111111 and 8 equals binary 1000, so the result is binary 1000 or 8 decimal.
4 AND 2=0	4 equals binary 100 and 2 equals binary 10, so the result is binary 0 because none of the bits in either argument match to give a 1 bit in the result.
4 OR 2=6	Binary 100 OR'd with binary 10 equals binary 110, or 6 decimal.
10 OR 10=10	Binary 1010 OR'd with binary 1010 equals binary 1010, or 10 decimal.
-1 OR -2=-1	Binary 1111111111111111 (-1) OR'd with binary 1111111111111110 (-2) equals binary 1111111111111111, or -1.
NOT 0=-1	The bit complement of binary 0 to 16 places is sixteen ones (1111111111111111) or -1. Also NOT -1=0.
NOT X	NOT X is equal to -(X+1). This is because to form the sixteen bit two's complement of the number, you take the bit (one's) complement and add one.
NOT 1=-2	The sixteen bit complement of 1 is 1111111111111110, which is equal to -(1+1) or -2.

A typical use of the bitwise operators is to test bits set in the ALTAIR's inport ports which reflect the state of some external device.

Bit position 7 is the most significant bit of a byte, while position 0 is the least significant.

For instance, suppose bit 1 of I/O port 5 is 0 when the door to Room X is closed, and 1 if the door is open. The following program will print "Intruder Alert" if the door is opened:

`10 IF NOT (INP(5) AND 2) THEN 10`	This line will execute over and over until bit 1 (masked or selected by the 2) becomes a 1. When that happens, we go to line 20 .
`20 PRINT "INTRUDER ALERT"`	Line 20 will output "INTRUDER ALERT".

However, we can replace statement 10 with a "WAIT" statement, which has exactly the same effect.

`10 WAIT 5,2`	This line delays the execution of the next statement in the program until bit 1 of I/O port 5 becomes 1. The WAIT is much faster than the equivalent IF statement and also takes less bytes of program storage.

The ALTAIR's sense switches may also be used as an input device by the INP function. The program below prints out any changes in the sense switches.

```
10 A=300:REM SET A TO A VALUE THAT WILL FORCE PRINTING
20 J=INP(255):IF J=A THEN 20
30 PRINT J;:A=J:GOTO 20
```

The following is another useful way of using relational operators:

`125 A=-(B>C)*B-(B<=C)*C`	This statement will set the variable A to MAX(B,C) = the larger of the two variables B and C.

STATEMENTS

Note: In the following description of statements, an argument of V or W denotes a numeric variable, X denotes a numeric expression, X$ denotes a string expression and an I or J denotes an expression that is truncated to an integer before the statement is executed. Truncation means that any fractional part of the number is lost, e.g. 3.9 becomes 3, 4.01 becomes 4.

An expression is a series of variables, operators, function calls and constants which after the operations and function calls are performed using the precedence rules, evaluates to a numeric or string value.

A constant is either a number (3.14) or a string literal ("FOO").

NAME	EXAMPLE	PURPOSE/USE
DATA	`10 DATA 1,3,-1E3,.04`	Specifies data, read from left to right. Information appears in data statements in the same order as it will be read in the program. IN THE 4K VERSION OF BASIC, DATA STATEMENTS MUST BE THE FIRST STATEMENTS ON A LINE. Expressions may also appear in the 4K version data statements.
	`20 DATA " FOO",ZOO`	*(8K Version)* Strings may be read from DATA statements. If you want the string to contain leading spaces (blanks), colons (:) or commas (,), you must enclose the string in double quotes. It is impossible to have a double quote within string data or a string literal. (""MITS"" is illegal)
DEF	`100 DEF FNA(V)=V/B+C`	*(8K Version)* The user can define functions like the built-in functions (SQR, SGN, ABS, etc.) through the use of the DEF statement. The name of the function is "FN" followed by any legal variable name, for example: FNX, FNJ7, FNKO, FNR2. User defined functions are restricted to one line. A function may be defined to be any expression, but may only have one argument. In the example B & C are variables that are used in the program. Executing the DEF statement defines the function. User defined functions can be redefined by executing another DEF statement for the same function. User defined string functions are not allowed. "V" is called the dummy variable.
	`110 Z=FNA(3)`	Execution of this statement following the above would cause Z to be set to 3/B+C, but the value of V would be unchanged.
DIM	`113 DIM A(3),B(10)`	Allocates space for matrices. All matrix elements are set to zero by the DIM statement.
	`114 DIM R3(5,5),D$(2,2,2)`	*(8K Version)* Matrices can have more than one dimension. Up to 255 dimensions are allowed, but due to the restriction of 72 characters per line the practical maximum is about 34 dimensions.
	`115 DIM Q1(N),Z(2*I)`	Matrices can be dimensioned dynamically during program execution. If a matrix is not explicitly dimensioned with a DIM statement, it is assumed to be a single dimensioned matrix of whose single subscript

117 A(8)=4 — may range from 0 to 10 (eleven elements). If this statement was encountered before a DIM statement for A was found in the program, it would be as if a DIM A(10) had been executed previous to the execution of line 117. All subscripts start at zero (0), which means that DIM X(100) really allocates 101 matrix elements.

END

999 END — Terminates program execution without printing a BREAK message. (see STOP) CONT after an END statement causes execution to resume at the statement after the END statement. END can be used anywhere in the program, and is optional.

FOR

300 FOR V=1 TO 9.3 STEP .6 — (see NEXT statement) V is set equal to the value of the expression following the equal sign, in this case 1. This value is called the initial value. Then the statements between FOR and NEXT are executed. The final value is the value of the expression following the TO. The step is the value of the expression following STEP. When the NEXT statement is encountered, the step is added to the variable.

310 FOR V=1 TO 9.3 — If no STEP was specified, it is assumed to be one. If the step is positive and the new value of the variable is <= the final value (9.3 in this example), or the step value is negative and the new value of the variable is => the final value, then the first statement following the FOR statement is executed. Otherwise, the statement following the NEXT statement is executed. All FOR loops execute the statements between the FOR and the NEXT at least once, even in cases like FOR V=1 TO 0.

315 FOR V=10*N TO 3.4/Q STEP SQR(R) — Note that expressions (formulas) may be used for the initial, final and step values in a FOR loop. The values of the expressions are computed only once, before the body of the FOR....NEXT loop is executed.

320 FOR V=9 TO 1 STEP -1 — When the statement after the NEXT is executed, the loop variable is never equal to the final value, but is equal to whatever value caused the FOR...NEXT loop to terminate. The statements between the FOR and its corresponding NEXT in both examples above (310 & 320) would be executed 9 times.

330 FOR W=1 TO 10: FOR W=1 TO :NEXT W:NEXT W — Error: do not use nested FOR...NEXT loops with the same index variable. FOR loop nesting is limited only by the available memory. (see Appendix D)

GOTO

50 GOTO 100 — Branches to the statement specified.

GOSUB

10 GOSUB 910 — Branches to the specified statement (910) until a RETURN is encountered; when a branch is then made to the statement after the GOSUB. GOSUB nesting is limited only by the available memory. (see Appendix D)

IF...GOTO

32 IF X<=Y+23.4 GOTO 92 — *(8K Version)* Equivalent to IF...THEN, except that IF...GOTO must be followed by a line number, while IF...THEN can be followed by either a line number or another statement.

IF...THEN

IF X<10 THEN 5 — Branches to specified statement if the relation is True.

20 IF X<0 THEN PRINT "X LESS THAN 0" — Executes all of the statements on the remainder of the line after the THEN if the relation is True.

25 IF X=5 THEN 50:Z=A — WARNING. The "Z=A" will never be executed because if the relation is true, BASIC will branch to line 50. If the relation is false Basic will proceed to the line after line 25.

26 IF X<0 THEN PRINT "ERROR, X NEGATIVE": GOTO 350 — In this example, if X is less than 0, the PRINT statement will be executed and then the GOTO statement will branch to line 350. If the X was 0 or positive, BASIC will proceed to execute the lines after line 26.

INPUT `3 INPUT V,W,W2` — Requests data from the terminal (to be typed in). Each value must be separated from the preceeding value by a comma (,). The last value typed should be followed by a carriage return. A "?" is typed as a prompt character. In the 4K version, a value typed in as a response to an INPUT statement may be a formula, such as 2*SIN(.16)-3. However, in the 8K version, only constants may be typed in as a response to an INPUT statement, such as 4.5E-3 or "CAT". If more data was requested in an INPUT statement than was typed in, a "??" is printed and the rest of the data should be typed in. If more data was typed in than was requested, the extra data will be ignored. The 8K version will print the warning "EXTRA IGNORED" when this happens. The 4K version will not print a warning message. *(8K Version)* Strings must be input in the same format as they are specified in DATA statements.

`5 INPUT "VALUE";V` — *(8K Version)* Optionally types a prompt string ("VALUE") before requesting data from the terminal. If carriage return is typed to an input statement, BASIC returns to command mode. Typing CONT after an INPUT command has been interrupted will cause execution to resume at the INPUT statement.

LET `300 LET W=X`, `310 V=5.1` — Assigns a value to a variable. "LET" is optional.

NEXT `340 NEXT V`, `345 NEXT` — Marks the end of a FOR loop. *(8K Version)* If no variable is given, matches the most recent FOR loop.

`350 NEXT V,W` — *(8K Version)* A single NEXT may be used to match multiple FOR statements. Equivalent to NEXT V:NEXT W.

ON...GOTO

`100 ON I GOTO 10,20,30,40` — *(8K Version)* Branches to the line indicated by the I'th number after the GOTO. That is:

IF I=1, THEN GOTO LINE 10
IF I=2, THEN GOTO LINE 20
IF I=3, THEN GOTO LINE 30
IF I=4, THEN GOTO LINE 40.

If I=0 or I attempts to select a non-existent line (>=5 in this case), the statement after the ON statement is executed. However, if I is >255 or <0, an FC error message will result. As many line numbers as will fit on a line can follow an ON...GOTO.

`105 ON SGN(X)+2 GOTO 40,50,60` — This statement will branch to line 40 if the expression X is less than zero, to line 50 if it equals zero, and to line 60 if it is greater than zero.

ON...GOSUB

`110 ON I GOSUB 50,60` — *(8K Version)* Identical to "ON...GOTO", except that a subroutine call (GOSUB) is executed instead of a GOTO. RETURN from the GOSUB branches to the statement after the ON...GOSUB.

OUT `355 OUT I,J` — *(8K Version)* Sends the byte J to the output port I. Both I & J must be >=0 and <=255.

POKE `357 POKE I,J` — *(8K Version)* The POKE statement stores the byte specified by its second argument (J) into the location given by its first argument (I). The byte to be stored must be =>0 and <=255, or an FC error will occur. The address (I) must be =>0 and <=32767, or an FC error will result. Careless use of the POKE statement will probably cause you to "poke" BASIC to death; that is, the machine will hang, and you will have to reload BASIC and will lose any program you had typed in. A POKE to a non-existent memory location is harmless. One of the main uses of POKE is to pass arguments to machine language subroutines. (see Appendix J) You could also use PEEK and POKE to write a memory diagnostic or an assembler in BASIC.

PRINT

```
360 PRINT X,Y;Z
370 PRINT
380 PRINT X,Y;
390 PRINT "VALUE IS";A
400 PRINT A2,B,
```

Prints the value of expressions on the terminal. If the list of values to be printed out does not end with a comma (,) or a semicolon (;), then a carriage return/line feed is executed after all the values have been printed. Strings enclosed in quotes (") may also be printed. If a semicolon separates two expressions in the list, their values are printed next to each other. If a comma appears after an

		expression in the list, and the print head is at print position 56 or more, then a carriage return/line feed is executed. If the print head is before print position 56, then spaces are printed until the carriage is at the beginning of the next 14 column field (until the carriage is at column 14, 28, 42 or 56...). If there is no list of expressions to be printed, as in line 370 of the examples, then a carriage return/line feed is executed.
	410 PRINT MID$(A$,2);	*(8K Version)* String expressions may be printed.
READ	450 READ V,W	Reads data into specified variables from a DATA statement. The first piece of data read will be the first piece of data listed in the first DATA statement of the program. The second piece of data read will be the second piece listed in the first DATA statement, and so on. When all of the data have been read from the first DATA statement, the next piece of data to be read will be the first piece listed in the second DATA statement of the program. Attempting to read more data than there is in all the DATA statements in a program will cause an OD (out of data) error. In the 4K version, an SN error from a READ statement can mean the data it was attempting to read from a DATA statement was improperly formatted. In the 8K version, the line number given in the SN error will refer to the line number where the error actually is located.
REM	500 REM NOW SET V=0	Allows the programmer to put comments in his program. REM statements are not executed, but can be branched to. A REM statement is terminated by end of line, but not by a ":".
	505 REM SET V=0: V=0	In this case the V=0 will never be executed by BASIC.
	506 V=0: REM SET V=0	In this case V=0 will be executed
RESTORE	510 RESTORE	Allows the re-reading of DATA statements. After a RESTORE, the next piece of data read will be the first piece listed in the first DATA statement of the program. The second piece of data read will be the second piece listed in the first DATA statement, and so on as in a normal READ operation.

RETURN	50 RETURN	Causes a subroutine to return to the statement after the most recently executed GOSUB.
STOP	9000 STOP	Causes a program to stop execution and to enter command mode. *(8K Version)* Prints BREAK IN LINE 9000. (as per this example) CONT after a STOP branches to the statement following the STOP.
WAIT	805 WAIT I,J,K 806 WAIT I,J	*(8K Version)* This statement reads the status of input port I, exclusive OR's K with the status, and then AND's the result with J until a non-zero result is obtained. Execution of the program continues at the statement following the WAIT statement. If the WAIT statement only has two arguments, K is assumed to be zero. If you are waiting for a bit to become zero, there should be a one in the corresponding position of K. I, J and K must be =>0 and <=255.

4K INTRINSIC FUNCTIONS

ABS(X)	120 PRINT ABS(X)	Gives the absolute value of the expression X. ABS returns X if X>=0, -X otherwise.
INT(X)	140 PRINT INT(X)	Returns the largest integer less than or equal to its argument X. For example: INT(.23)=0, INT(7)=7, INT(-.1)=-1, INT(-2)= -2, INT(1.1)=1. The following would round X to D decimal places: INT(X*10↑D+.5)/10↑D
RND(X)	170 PRINT RND(X)	Generates a random number between 0 and 1. The argument X controls the generation of random numbers as follows: X<0 starts a new sequence of random numbers using X. Calling RND with the same X starts the same random number sequence. X=0 gives the last random number generated. Repeated calls to RND(0) will always return the same random number. X>0 generates a new random number between 0 and 1. Note that (B-A)*RND(1)+A will generate a random number between A & B.

SGN(X)	230 PRINT SGN(X)	Gives 1 if X>0, 0 if X=0, and -1 if X<0.
SIN(X)	190 PRINT SIN(X)	Gives the sine of the expression X. X is interpreted as being in radians. Note: COS (X)=SIN(X+3.14159/2) and that 1 Radian =180/PI degrees=57.2958 degrees; so that the sine of X degrees= SIN(X/57.2958).
SQR(X)	180 PRINT SQR(X)	Gives the square root of the argument X. An FC error will occur if X is less than zero.
TAB(I)	240 PRINT TAB(I)	Spaces to the specified print position (column) on the terminal. May be used only in PRINT statements. Zero is the leftmost column on the terminal, 71 the rightmost. If the carriage is beyond position I, then no printing is done. I must be =>0 and <=255.
USR(I)	200 PRINT USR(I)	Calls the user's machine language subroutine with the argument I. See POKE, PEEK and ~~Appendix J.~~

8K FUNCTIONS *(Includes all those listed under 4K INTRINSIC FUNCTIONS plus the following in addition.)*

ATN(X)	210 PRINT ATN(X)	Gives the arctangent of the argument X. The result is returned in radians and ranges from -PI/2 to PI/2. (PI/2=1.5708)
COS(X)	200 PRINT COS(X)	Gives the cosine of the expression X. X is interpreted as being in radians.
EXP(X)	150 PRINT EXP(X)	Gives the constant "E" (2.71828) raised to the power X. (E↑X) The maximum argument that can be passed to EXP without overflow occuring is 87.3365.
FRE(X)	270 PRINT FRE(0)	Gives the number of memory bytes currently unused by BASIC. Memory allocated for STRING space is not included in the count returned by FRE. To find the number of free bytes in STRING space, call FRE with a STRING argument. (see FRE under STRING FUNCTIONS)
INP(I)	265 PRINT INP(I)	Gives the status of (reads a byte from) input port I. Result is =>0 and <=255.

LOG(X)	160 PRINT LOG(X)	Gives the natural (Base E) logarithm of its argument X. To obtain the Base Y logarithm of X use the formula LOG(X)/LOG(Y). Example: The base 10 (common) log of 7 = LOG(7)/ LOG(10).
PEEK	356 PRINT PEEK(I)	The PEEK function returns the contents of memory address I. The value returned will be =>0 and <=255. If I is >32767 or <0, an FC error will occur. An attempt to read a non-existent memory address will return 255. (see POKE statement)
POS(I)	260 PRINT POS(I)	Gives the current position of the terminal print head (or cursor on CRT's). The leftmost character position on the terminal is position zero and the rightmost is 71.
SPC(I)	250 PRINT SPC(I)	Prints I space (or blank) characters on the terminal. May be used only in a PRINT statement. X must be =>0 and <=255 or an FC error will result.
TAN(X)	200 PRINT TAN(X)	Gives the tangent of the expression X. X is interpreted as being in radians.

STRINGS *(8K Version Only)*

1) A string may be from 0 to 255 characters in length. All string variables end in a dollar sign ($); for example, A$, B9$, K$, HELLO$.

2) String matrices may be dimensioned exactly like numeric matrices. For instance, DIM A$(10,10) creates a string matrix of 121 elements, eleven rows by eleven columns (rows 0 to 10 and columns 0 to 10). Each string matrix element is a complete string, which can be up to 255 characters in length.

3) The total number of characters in use in strings at any time during program execution cannot exceed the amount of string space, or an OS error will result. At initialization, you should set up string space so that it can contain the maximum number of characters which can be used by strings at any one time during program execution.

NAME	EXAMPLE	PURPOSE/USE
DIM	25 DIM A$(10,10)	Allocates space for a pointer and length for each element of a string matrix. No string space is allocated. See Appendix D.

LET	27 LET A$="FOO"+V$	Assigns the value of a string expression to a string variable. LET is optional.
= > < <= >= <>		String comparison operators. Comparison is made on the basis of ASCII codes, a character at a time until a difference is found. If during the comparison of two strings, the end of one is reached, the shorter string is considered smaller. Note that "A " is greater than "A" since trailing spaces are significant.
+	30 LET Z$=R$+Q$	String concatentation. The resulting string must be less than 256 characters in length or an LS error will occur.
INPUT	40 INPUT X$	Reads a string from the user's terminal. String does not have to be quoted; but if not, leading blanks will be ignored and the string will be terminated on a "," or ":" character.
READ	50 READ X$	Reads a string from DATA statements within the program. Strings do not have to be quoted; but if they are not, they are terminated on a "," or ":" character or end of line and leading spaces are ignored. See DATA for the format of string data.
PRINT	60 PRINT X$ 70 PRINT "FOO"+A$	Prints the string expression on the user's terminal.

STRING FUNCTIONS *(8K Version Only)*

ASC(X$)	300 PRINT ASC(X$)	Returns the ASCII numeric value of the first character of the string expression X$. See Appendix K for an ASCII/number conversion table. An FC error will occur if X$ is the null string.
CHR$(I)	275 PRINT CHR$(I)	Returns a one character string whose single character is the ASCII equivalent of the value of the argument (I) which must be =>0 and <=255. See Appendix K.
FRE(X$)	272 PRINT FRE("")	When called with a string argument, FRE gives the number of free bytes in string space.
LEFT$(X$,I)	310 PRINT LEFT$(X$,I)	Gives the leftmost I characters of the string expression X$. If I<=0 or >255 an FC error occurs.

LEN(X$)	220 PRINT LEN(X$)	Gives the length of the string expression X$ in characters (bytes). Non-printing characters and blanks are counted as part of the length.
MID$(X$,I)	330 PRINT MID$(X$,I)	MID$ called with two arguments returns characters from the string expression X$ starting at character position I. If I>LEN(I$), then MID$ returns a null (zero length) string. If I<=0 or >255, an FC error occurs.
MID$(X$,I,J)	340 PRINT MID$(X$,I,J)	MID$ called with three arguments returns a string expression composed of the characters of the string expression X$ starting at the Ith character for J characters. If I>LEN(X$), MID$ returns a null string. If I or J <=0 or >255, an FC error occurs. If J specifies more characters than are left in the string, all characters from the Ith on are returned.
RIGHT$(X$,I)	320 PRINT RIGHT$(X$,I)	Gives the rightmost I characters of the string expression X$. When I<=0 or >255 an FC error will occur. If I>=LEN(X$) then RIGHT$ returns all of X$.
STR$(X)	290 PRINT STR$(X)	Gives a string which is the character representation of the numeric expression X. For instance, STR$(3.1)=" 3.1".
VAL(X$)	280 PRINT VAL(X$)	Returns the string expression X$ converted to a number. For instance, VAL("3.1")=3.1. If the first non-space character of the string is not a plus (+) or minus (-) sign, a digit or a decimal point (.) then zero will be returned.

SPECIAL CHARACTERS

CHARACTER	USE
@	Erases current line being typed, and types a carriage return/line feed. An "@" is usually a shift/P.
←	*(backarrow or underline)* Erases last character typed. If no more characters are left on the line, types a carriage return/line feed. "←" is usually a shift/O.

CARRIAGE RETURN	A carriage return must end every line typed in. Returns print head or CRT cursor to the first position (leftmost) on line. A line feed is always executed after a carriage return.
CONTROL/C	Interrupts execution of a program or a list command. Control/C has effect when a statement finishes execution, or in the case of interrupting a LIST command, when a complete line has finished printing. In both cases a return is made to BASIC's command level and OK is typed. *(8K Version)* Prints "BREAK IN LINE XXXX" , where XXXX is the line number of the next statement to be executed.
: (colon)	A colon is used to separate statements on a line. Colons may be used in direct and indirect statements. The only limit on the number of statements per line is the line length. It is not possible to GOTO or GOSUB to the middle of a line.

(8K Version Only)

CONTROL/O	Typing a Control/O once causes BASIC to suppress all output until a return is made to command level, an input statement is encountered, another control/O is typed, or an error occurs.
?	Question marks are equivalent to PRINT. For instance, ? 2+2 is equivalent to PRINT 2+2. Question marks can also be used in indirect statements. 10 ? X, when listed will be typed as 10 PRINT X.

MISCELLANEOUS

1) To read in a paper tape with a program on it (8K Version), type a control/O and feed in tape. There will be no printing as the tape is read in. Type control/O again when the tape is through. Alternatively, set nulls=0 and feed in the paper tape, and when done reset nulls to the appropriate setting for your terminal. Each line must be followed by two rubouts, or any other non-printing character. If there are lines without line numbers (direct commands) the ALTAIR will fall behind the input coming from paper tape, so this in not recommending.

Using null in this fashion will produce a listing of your tape in the 8K version (use control/O method if you don't want a listing). The null method is the only way to read in a tape in the 4K version.

To read in a paper tape of a program in the 4K version, set the number of nulls typed on carriage return/line feed to zero by patching location 46 (octal) to be a 1. Feed in the paper tape. When

the tape has finished reading, stop the CPU and repatch location 46 to be the appropriate number of null characters (usually 0, so deposit a 1). When the tape is finished, BASIC will print SN ERROR because of the "OK" at the end of the tape.

2) To punch a paper tape of a program, set the number of nulls to 3 for 110 BAUD terminals (Teletypes) and 6 for 300 BAUD terminals. Then, type LIST; but, do not type a carriage return. Now, turn on the terminal's paper tape punch. Put the terminal on local and hold down the Repeat, Control, Shift and P keys at the same time. Stop after you have punched about a 6 to 8 inch leader of nulls. These nulls will be ignored by BASIC when the paper tape is read in. Put the terminal back on line. Now hit carriage return. After the program has finished punching, put some trailer on the paper tape by holding down the same four keys as before, with the terminal on local. After you have punched about a six inch trailer, tear off the paper tape and save for later use as desired.

3) Restarting BASIC at location zero (by toggling STOP, Examine location 0, and RUN) will cause BASIC to return to command level and type "OK". However, typing Control/C is preferred because Control/C is guaranteed not to leave garbage on the stack and in variables, and a Control C'd program may be continued. (see CONT command)

4) The maximum line length is 72 characters.** If you attempt to type too many characters into a line, a bell (ASCII 7) is executed, and the character you typed in will not be echoed. At this point you can either type backarrow to delete part of the line, or at-sign to delete the whole line. The character you typed which caused BASIC to type the bell is not inserted in the line as it occupies the character position one beyond the end of the line.

*CLEAR	CLEAR	Deletes all variables.
	CLEAR X	*(8K Version)* Deletes all variables. When used with an argument "X", sets the amount of space to be allocated for use by string variables to the number indicated by its argument "X".
	10 CLEAR 50	*(8K Version)* Same as above; but, may be used at the beginning of a program to set the exact amount of string space needed, leaving a maximum amount of memory for the program itself.
		NOTE: If no argument is given, the string space is set at 200 by default. An OM error will occur if an attempt is made to allocate more string space than there is available memory.

**For inputting only.

APPENDIX A

HOW TO LOAD BASIC

When the ALTAIR is first turned on, there is random garbage in its memory. BASIC is supplied on a paper tape or audio cassette. Somehow the information on the paper tape or cassette must be transfered into the computer. Programs that perform this type of information transfer are called loaders.

Since initially there is nothing of use in memory; you must toggle in, using the switches on the front panel, a 20 instruction bootstrap loader. This loader will then load BASIC.

To load BASIC follow these steps:

1) Turn the ALTAIR on.

2) Raise the STOP switch and RESET switch simultaneously.

3) Turn your terminal (such as a Teletype) to LINE.

Because the instructions must be toggled in via the switches on the front panel, it is rather inconvenient to specify the positions of each switch as "up" or "down". Therefore, the switches are arranged in groups of 3 as indicated by the broken lines below switches 0 through 15. To specify the positions of each switch, we use the numbers 0 through 7 as shown below:

3 SWITCH GROUP

LEFTMOST	MIDDLE	RIGHTMOST	OCTAL NUMBER
Down	Down	Down	0
Down	Down	Up	1
Down	Up	Down	2
Down	Up	Up	3
Up	Down	Down	4
Up	Down	Up	5
Up	Up	Down	6
Up	Up	Up	7

So, to put the octal number 315 in switches 0 through 7, the switches would have the following positions:

7	6	5	4	3	2	1	0	← SWITCH
UP	UP	DOWN	DOWN	UP	UP	DOWN	UP	← POSITION
3			1			5		← OCTAL NO.

Note that switches 8 through 15 were not used. Switches 0 through 7 correspond to the switches labeled DATA on the front panel. A memory address would use all 16 switches.

The following program is the bootstrap loader for users loading from paper tape, and not using a REV 0 Serial I/O Board.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	022	
005	000	
006	333	
007	000	
010	017	
011	330	
012	333	
013	001	
014	275	
015	310	
016	055	
017	167	
020	300	
021	351	
022	003	
023	000	

The following 21 byte bootstrap loader is for users loading from a paper tape and using a REV 0 Serial I/O Board on which the update changing the flag bits has not been made. If the update has been made, use the above bootstrap loader.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	023	
005	000	
006	333	
007	000	
010	346	
011	040	
012	310	
013	333	
014	001	
015	275	
016	310	
017	055	
020	167	

OCTAL ADDRESS (cont.)	OCTAL DATA
021	300
022	351
023	003
024	000

The following bootstrap loader is for users with BASIC supplied on an audio cassette.

OCTAL ADDRESS	OCTAL DATA	
000	041	
001	175	
002	037	(for 8K; for 4K use 017)
003	061	
004	022	
005	000	
006	333	
007	006	
010	017	
011	330	
012	333	
013	007	
014	275	
015	310	
016	055	
017	167	
020	300	
021	351	
022	003	
023	000	

To load a bootstrap loader:

1) Put switches 0 through 15 in the down position.

2) Raise EXAMINE.

3) Put 041 (data for address 000) in switches 0 through 7.

4) Raise DEPOSIT.

5) Put the data for the next address in switches 0 through 7.

6) Depress DEPOSIT NEXT.

7) Repeat steps 5 & 6 until the entire loader is toggled in.

8) Put switches 0 through 15 in the down position.

9) Raise EXAMINE.

10) Check that lights D0 through D7 correspond with the data that should

be in address 000. A light on means the switch was up, a light off means the switch was down. So for address 000, lights D1 through D4 and lights D6 & D7 should be off, and lights D0 and D5 should be on.

If the correct value is there, go to step 13. If the value is wrong, continue with step 11.

11) Put the correct value in switches 0 through 7.

12) Raise DEPOSIT.

13) Depress EXAMINE NEXT.

14) Repeat steps 10 through 13, checking to see that the correct data is in each corresponding address for the entire loader.

15) If you encountered any mistakes while checking the loader, go back now and re-check the whole program to be sure it is corrected.

16) Put the tape of BASIC into the tape reader. Be sure the tape is positioned at the beginning of the leader. The leader is the section of tape at the beginning with 6 out of the 8 holes punched.

If you are loading from audio cassette, put the cassette in the recorder. Be sure the tape is fully rewound.

17) Put switches 0 through 15 in the down position.

18) Raise EXAMINE.

19) If you have connected to your terminal a REV 0 Serial I/O Board on which the update changing the flag bits has not been made, raise switch 14; if you are loading from an audio cassette, raise switch 15 also.

If you have a REV 0 Serial I/O Board which has been updated, or have a REV 1 I/O Board, switch 14 should remain down and switch 15 should be raised only if you are loading from audio cassette.

20) Turn on the tape reader and then depress RUN. Be sure RUN is depressed while the reader is still on the leader. Do not depress run before turning on the reader, since this may cause the tape to be read incorrectly.

If you are loading from a cassette, turn the cassette recorder to Play. Wait 15 seconds and then depress RUN.

21) Wait for the tape to be read in. This should take about 12 minutes for 8K BASIC and 6 minutes for 4K BASIC. It takes about 4 minutes to load 8K BASIC from cassette, and about 2 minutes for 4K BASIC.

Do not move the switches while the tape is being read in.

22) If a C or an O is printed on the terminal as the tape reads in, the tape has been mis-read and you should start over at step 1 on page 46.

23) When the tape finishes reading, BASIC should start up and print MEMORY SIZE?. See Appendix B for the initialization procedure.

24) If BASIC refuses to load from the Audio Cassette, the ACR Demodulator may need alignment. The flip side of the cassette contains 90 seconds of 125's (octal) which were recorded at the same tape speed as BASIC. Use the Input Test Program described on pages 22 and 28 of the ACR manual to perform the necessary alignment.

APPENDIX B

INITIALIZATION DIALOG

STARTING BASIC

Leave the sense switches as they were set for loading BASIC (Appendix A). After the initialization dialog is complete, and BASIC types OK, you are free to use the sense switches as an input device (I/O port 255).

After you have loaded BASIC, it will respond:

MEMORY SIZE?

If you type a carriage return to MEMORY SIZE?, BASIC will use all the contiguous memory upwards from location zero that it can find. BASIC will stop searching when it finds one byte of ROM or non-existent memory.
If you wish to allocate only part of the ALTAIR's memory to BASIC, type the number of bytes of memory you wish to allocate in decimal. This might be done, for instance, if you were using part of the memory for a machine language subroutine.
There are 4096 bytes of memory in a 4K system, and 8192 bytes in an 8K system.

BASIC will then ask:

TERMINAL WIDTH?	This is to set the output line width for PRINT statements only. Type in the number of characters for the line width for the particular terminal or other output device you are using. This may be any number from 1 to 255, depending on the terminal. If no answer is given (i.e. a carriage return is typed) the line width is set to 72 characters.

Now ALTAIR BASIC will enter a dialog which will allow you to delete some of the arithmetic functions. Deleting these functions will give more memory space to store your programs and variables. However, you will not be able to call the functions you delete. Attempting to do so will result in an FC error. The only way to restore a function that has been deleted is to reload BASIC.

The following is the dialog which will occur:

4K Version

WANT SIN?	Answer " Y " to retain SIN, SQR and RND. If you answer " N ", asks next question.
WANT SQR?	Answer " Y " to retain SQR and RND. If you answer " N ", asks next question.

WANT RND? — Answer " Y " to retain RND.
Answer " N " to delete RND.

8K Version

WANT SIN-COS-TAN-ATN? — Answer " Y " to retain all four of the functions, " N " to delete all four, or " A " to delete ATN only.

Now BASIC will type out:

```
XXXX BYTES FREE

ALTAIR BASIC VERSION 3.0
[FOUR-K VERSION]
      (or)
[EIGHT-K VERSION]

OK
```

"XXXX" is the number of bytes available for program, variables, matrix storage and the stack. It does not include string space.

You will now be ready to begin using ALTAIR BASIC.

APPENDIX C

ERROR MESSAGES

After an error occurs, BASIC returns to command level and types OK. Variable values and the program text remain intact, but the program can not be continued and all GOSUB and FOR context is lost.

When an error occurs in a direct statement, no line number is printed.

Format of error messages:

Direct Statement	?XX ERROR
Indirect Statement	?XX ERROR IN YYYYY

In both of the above examples, "XX" will be the error code. The "YYYYY" will be the line number where the error occured for the indirect statement.

The following are the possible error codes and their meanings:

ERROR CODE	MEANING
4K VERSION	
BS	Bad Subscript. An attempt was made to reference a matrix element which is outside the dimensions of the matrix. In the 8K version, this error can occur if the wrong number of dimensions are used in a matrix reference; for instance, LET A(1,1,1)=Z when A has been dimensioned DIM A(2,2).
DD	Double Dimension. After a matrix was dimensioned, another dimension statement for the same matrix was encountered. This error often occurs if a matrix has been given the default dimension 10 because a statement like A(I)=3 is encountered and then later in the program a DIM A(100) is found.
FC	Function Call error. The parameter passed to a math or string function was out of range. FC errors can occur due to:

a) a negative matrix subscript (LET A(-1)=0)

b) an unreasonably large matrix subscript (>32767)

c) LOG-negative or zero argument

d) SQR-negative argument

e) A↑B with A negative and B not an integer

f) a call to USR before the address of the machine language subroutine has been patched in

g) calls to MID$, LEFT$, RIGHT$, INP, OUT, WAIT, PEEK, POKE, TAB, SPC or ON...GOTO with an improper argument.

ID . Illegal Direct. You cannot use an INPUT or *(in 8K Version)* DEFFN statement as a direct command.

NF . NEXT without FOR. The variable in a NEXT statement corresponds to no previously executed FOR statement.

OD . Out of Data. A READ statement was executed but all of the DATA statements in the program have already been read. The program tried to read too much data or insufficient data was included in the program.

OM Out of Memory. Program too large, too many variables, too many FOR loops, too many GOSUB's, too complicated an expression or any combination of the above. (see Appendix D)

OV . Overflow. The result of a calculation was too large to be represented in BASIC's number format. If an underflow occurs, zero is given as the result and execution continues without any error message being printed.

SN . Syntax error. Missing parenthesis in an expression, illegal character in a line, incorrect punctuation, etc.

RG . RETURN without GOSUB. A RETURN statement was encountered without a previous GOSUB statement being executed.

US . Undefined Statement. An attempt was made to GOTO, GOSUB or THEN to a statement which does not exist.

/0 . Division by Zero.

8K VERSION *(Includes all of the previous codes in addition to the following.)*

CN . Continue error. Attempt to continue a program when none exists, an error occured, or after a new line was typed into the program.

LS . Long String. Attempt was made by use of the concatenation operator to create a string more than 255 characters long.

OS . Out of String Space. Save your program on paper tape or cassette, reload BASIC and allocate more string space or use smaller strings or less string variables.

ST . String Temporaries. A string expression was too complex. Break it into two or more shorter ones.

TM . Type Mismatch. The left hand side of an assignment statement was a numeric variable and the right hand side was a string, or vice versa; or, a function which expected a string argument was given a numeric one or vice versa.

UF . Undefined Function. Reference was made to a user defined function which had never been defined.

APPENDIX D

SPACE HINTS

In order to make your program smaller and save space, the following hints may be useful.

1) Use multiple statements per line. There is a small amount of overhead (5bytes) associated with each line in the program. Two of these five bytes contain the line number of the line in binary. This means that no matter how many digits you have in your line number (minimum line number is 0, maximum is 65529), it takes the same number of bytes. Putting as many statements as possible on a line will cut down on the number of bytes used by your program.

2) Delete all unnecessary spaces from your program. For instance:

```
10 PRINT X, Y, Z
```

uses three more bytes than

```
10 PRINTX,Y,Z
```

Note: All spaces between the line number and the first non-blank character are ignored.

3) Delete all REM statements. Each REM statement uses at least one byte plus the number of bytes in the comment text. For instance, the statement 130 REM THIS IS A COMMENT uses up 24 bytes of memory.
In the statement 140 X=X+Y: REM UPDATE SUM, the REM uses 14 bytes of memory including the colon before the REM.

4) Use variables instead of constants. Suppose you use the constant 3.14159 ten times in your program. If you insert a statement

```
10 P=3.14159
```

in the program, and use P instead of 3.14159 each time it is needed, you will save 40 bytes. This will also result in a speed improvement.

5) A program need not end with an END; so, an END statement at the end of a program may be deleted.

6) Reuse the same variables. If you have a variable T which is used to hold a temporary result in one part of the program and you need a temporary variable later in your program, use it again. Or, if you are asking the terminal user to give a YES or NO answer to two different questions at two different times during the execution of the program, use the same temporary variable A$ to store the reply.

7) Use GOSUB's to execute sections of program statements that perform identical actions.

8) If you are using the 8K version and don't need the features of the 8K version to run your program, consider using the 4K version instead. This will give you approximately 4.7K to work with in an 8K machine, as opposed to the 1.6K you have available in an 8K machine running the 8K version of BASIC.

9) Use the zero elements of matrices; for instance, A(0), B(0,X).

STORAGE ALLOCATION INFORMATION

Simple (non-matrix) numeric variables like V use 6 bytes; 2 for the variable name, and 4 for the value. Simple non-matrix string variables also use 6 bytes; 2 for the variable name, 2 for the length, and 2 for a pointer.

Matrix variables use a minimum of 12 bytes. Two bytes are used for the variable name, two for the size of the matrix, two for the number of dimensions and two for each dimension along with four bytes for each of the matrix elements.

String variables also use one byte of string space for each character in the string. This is true whether the string variable is a simple string variable like A$, or an element of a string matrix such as Q1$(5,2).

When a new function is defined by a DEF statement, 6 bytes are used to store the definition.

Reserved words such as FOR, GOTO or NOT, and the names of the intrinsic functions such as COS, INT and STR$ take up only one byte of program storage. All other characters in programs use one byte of program storage each.

When a program is being executed, space is dynamically allocated on the stack as follows:

1) Each active FOR...NEXT loop uses 16 bytes.

2) Each active GOSUB (one that has not returned yet) uses 6 bytes.

3) Each parenthesis encountered in an expression uses 4 bytes and each temporary result calculated in an expression uses 12 bytes.

APPENDIX E

SPEED HINTS

The hints below should improve the execution time of your BASIC program. Note that some of these hints are the same as those used to decrease the space used by your programs. This means that in many cases you can increase the efficiency of both the speed and size of your programs at the same time.

1) Delete all unnecessary spaces and REM's from the program. This may cause a small decrease in execution time because BASIC would otherwise have to ignore or skip over spaces and REM statements.

2) *THIS IS PROBABLY THE MOST IMPORTANT SPEED HINT BY A FACTOR OF 10.* Use variables instead of constants. It takes more time to convert a constant to its floating point representation than it does to fetch the value of a simple or matrix variable. This is especially important within FOR...NEXT loops or other code that is executed repeatedly.

3) Variables which are encountered first during the execution of a BASIC program are allocated at the start of the variable table. This means that a statement such as 5 A=0:B=A:C=A, will place A first, B second, and C third in the symbol table (assuming line 5 is the first statement executed in the program). Later in the program, when BASIC finds a reference to the variable A, it will search only one entry in the symbol table to find A, two entries to find B and three entries to find C, etc.

4) *(8K Version)* NEXT statements without the index variable. NEXT is somewhat faster than NEXT I because no check is made to see if the variable specified in the NEXT is the same as the variable in the most recent FOR statement.

5) Use the 8K version instead of the 4K version. The 8K version is about 40% faster than the 4K due to improvements in the floating point arithmetic routines.

6) The math functions in the 8K version are much faster than their counterparts simulated in the 4K version. (see Appendix G)

APPENDIX F

DERIVED FUNCTIONS

The following functions, while not intrinsic to ALTAIR BASIC, can be calculated using the existing BASIC functions.

FUNCTION	FUNCTION EXPRESSED IN TERMS OF BASIC FUNCTIONS
SECANT	SEC(X) = 1/COS(X)
COSECANT	CSC(X) = 1/SIN(X)
COTANGENT	COT(X) = 1/TAN(X)
INVERSE SINE	ARCSIN(X) = ATN(X/SQR(-X*X+1))
INVERSE COSINE	ARCCOS(X) = -ATN(X/SQR(-X*X+1))+1.5708
INVERSE SECANT	ARCSEC(X) = ATN(SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COSECANT	ARCCSC(X) = ATN(1/SQR(X*X-1))+(SGN(X)-1)*1.5708
INVERSE COTANGENT	ARCCOT(X) = -ATN(X)+1.5708
HYPERBOLIC SINE	SINH(X) = (EXP(X)-EXP(-X))/2
HYPERBOLIC COSINE	COSH(X) = (EXP(X)+EXP(-X))/2
HYPERBOLIC TANGENT	TANH(X) = -EXP(-X)/(EXP(X)+EXP(-X))*2+1
HYPERBOLIC SECANT	SECH(X) = 2/(EXP(X)+EXP(-X))
HYPERBOLIC COSECANT	CSCH(X) = 2/(EXP(X)-EXP(-X))
HYPERBOLIC COTANGENT	COTH(X) = EXP(-X)/(EXP(X)-EXP(-X))*2+1
INVERSE HYPERBOLIC SINE	ARGSINH(X) = LOG(X+SQR(X*X+1))
INVERSE HYPERBOLIC COSINE	ARGCOSH(X) = LOG(X+SQR(X*X-1))
INVERSE HYPERBOLIC TANGENT	ARGTANH(X) = LOG((1+X)/(1-X))/2
INVERSE HYPERBOLIC SECANT	ARGSECH(X) = LOG((SQR(-X*X+1)+1)/X)
INVERSE HYPERBOLIC COSECANT	ARGCSCH(X) = LOG((SGN(X)*SQR(X*X+1)+1)/X)
INVERSE HYPERBOLIC COTANGENT	ARGCOTH(X) = LOG((X+1)/(X-1))/2

APPENDIX G

SIMULATED MATH FUNCTIONS

The following subroutines are intended for 4K BASIC users who want to use the transcendental functions not built into 4K BASIC. The corresponding routines for these functions in the 8K version are much faster and more accurate. The REM statements in these subroutines are given for documentation purposes only, and should not be typed in because they take up a large amount of memory.

The following are the subroutine calls and their 8K equivalents:

8K EQUIVALENT	SUBROUTINE CALL
P9=X9↑Y9	GOSUB 60030
L9=LOG(X9)	GOSUB 60090
E9=EXP(X9)	GOSUB 60160
C9=COS(X9)	GOSUB 60240
T9=TAN(X9)	GOSUB 60280
A9=ATN(X9)	GOSUB 60310

The unneeded subroutines should not be typed in. Please note which variables are used by each subroutine. Also note that TAN and COS require that the SIN function be retained when BASIC is loaded and initialized.

```
60000 REM EXPONENTIATION: P9=X9↑Y9
60010 REM NEED: EXP, LOG
60020 REM VARIABLES USED: A9,B9,C9,E9,L9,P9,X9,Y9
60030 P9=1 : E9=0 : IF Y9=0 THEN RETURN
60040 IF X9<0 THEN IF INT(Y9)=Y9 THEN P9=1-2*Y9+4*INT(Y9/2) : X9=-X9
60050 IF X9<>0 THEN GOSUB 60090 : X9=Y9*L9 : GOSUB 60160
60060 P9=P9*E9 : RETURN
60070 REM NATURAL LOGARITHM: L9=LOG(X9)
60080 REM VARIABLES USED: A9,B9,C9,E9,L9,X9
60090 E9=0 : IF X9<=0 THEN PRINT "LOG FC ERROR"; : STOP
60095 A9=1 : B9=2 : C9=.5 : REM THIS WILL SPEED UP THE FOLLOWING
60100 IF X9>=A9 THEN X9=C9*X9 : E9=E9+A9 : GOTO 60100
60110 IF X9<C9 THEN X9=B9*X9 : E9=E9-A9 : GOTO 60110
60120 X9=(X9-.707107)/(X9+.707107) : L9=X9*X9
60130 L9=(((.598979*L9+.961471)*L9+2.88539)*X9+E9-.5)*.693147
60135 RETURN
60140 REM EXPONENTIAL: E9=EXP(X9)
60150 REM VARIABLES USED: A9,E9,L9,X9
60160 L9=INT(1.4427*X9)+1 : IF L9<127 THEN 60180
60170 IF X9>0 THEN PRINT "EXP OV ERROR"; : STOP
60175 E9=0 : RETURN
60180 E9=.693147*L9-X9 : A9=1.32988E-3-1.41316E-4*E9
60190 A9=((A9*E9-8.30136E-3)*E9+4.16574E-2)*E9
60195 E9=(((A9-.166665)*E9+.5)*E9-1)*E9+1 : A9=2
60197 IF L9<=0 THEN A9=.5 : L9=-L9 : IF L9=0 THEN RETURN
```



```
60200 FOR X9=1 TO L9 : E9=A9*E9 : NEXT X9 : RETURN
60210 REM COSINE: C9=COS(X9)
60220 REM N.B. SIN MUST BE RETAINED AT LOAD-TIME
60230 REM VARIABLES USED: C9,X9
60240 C9=SIN(X9+1.5708) : RETURN
60250 REM TANGENT: T9=TAN(X9)
60260 REM NEEDS COS. (SIN MUST BE RETAINED AT LOAD-TIME)
60270 REM VARIABLES USED: C9,T9,X9
60280 GOSUB 60240 : T9=SIN(X9)/C9 : RETURN
60290 REM ARCTANGENT: A9=ATN(X9)
60300 REM VARIABLES USED: A9,B9,C9,T9,X9
60310 T9=SGN(X9): X9=ABS(X9): C9=0 : IF X9>1 THEN C9=1 : X9=1/X9
60320 A9=X9*X9 : B9=((2.86623E-3*A9-1.61657E-2)*A9+4.29096E-2)*A9
60330 B9=((((B9-7.5289E-2)*A9+.106563)*A9-.142089)*A9+.199936)*A9
60340 A9=((B9-.333332)*A9+1)*X9 : IF C9=1 THEN A9=1.5708-A9
60350 A9=T9*A9 : RETURN
```

APPENDIX H

CONVERTING BASIC PROGRAMS NOT WRITTEN FOR THE ALTAIR

Though implementations of BASIC on different computers are in many ways similar, there are some incompatibilites which you should watch for if you are planning to convert some BASIC programs that were not written for the ALTAIR.

1) Matrix subscripts. Some BASICs use " [" and "] " to denote matrix subscripts. ALTAIR BASIC uses " (" and ") ".

2) Strings. A number of BASICs force you to dimension (declare) the length of strings before you use them. You should remove all dimension statements of this type from the program. In some of these BASICs, a declaration of the form DIM A$(I,J) declares a string matrix of J elements each of which has a length I. Convert DIM statements of this type to equivalent ones in ALTAIR BASIC: DIM A$(J).

ALTAIR BASIC uses " + " for string concatenation, not " , " or " & ".

ALTAIR BASIC uses LEFT$, RIGHT$ and MID$ to take substrings of strings. Other BASICs use A$(I) to access the Ith character of the string A$, and A$(I,J) to take a substring of A$ from character position I to character position J. Convert as follows:

OLD	NEW
A$(I)	MID$(A$,I,1)
A$(I,J)	MID$(A$,I,J-I+1)

This assumes that the reference to a substring of A$ is in an expression or is on the right side of an assignment. If the reference to A$ is on the left hand side of an assignment, and X$ is the string expression used to replace characters in A$, convert as follows:

OLD	NEW
A$(I)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,I+1)
A$(I,J)=X$	A$=LEFT$(A$,I-1)+X$+MID$(A$,J+1)

3) Multiple assignments. Some BASICs allow statements of the form: 500 LET B=C=0. This statement would set the variables B & C to zero.

In 8K ALTAIR BASIC this has an entirely different effect. All the " ='s " to the right of the first one would be interpreted as logical comparison operators. This would set the variable B to -1 if C equaled 0. If C did not equal 0, B would be set to 0. The easiest way to convert statements like this one is to rewrite them as follows:


```
500 C=0:B=C.
```

4) Some BASICs use " \ " instead of " : " to delimit multiple statements per line. Change the " \'s " to " :'s " in the program.

5) Paper tapes punched by other BASICs may have no nulls at the end of each line, instead of the three per line recommended for use with ALTAIR BASIC.

To get around this, try to use the tape feed control on the Teletype to stop the tape from reading as soon as ALTAIR BASIC types a carriage return at the end of the line. Wait a second, and then continue feeding in the tape.

When you have finished reading in the paper tape of the program, be sure to punch a new tape in ALTAIR BASIC's format. This will save you from having to repeat this process a second time.

6) Programs which use the MAT functions available in some BASICs will have to be re-written using FOR...NEXT loops to perform the appropriate operations.

APPENDIX I

start op 8080 versie.

USING THE ACR INTERFACE

NOTE: The cassette features, CLOAD and CSAVE, are only present in 8K BASICs which are distributed on cassette. 8K BASIC on paper tape will give the user about 130 more bytes of free memory, but it will not recognize the CLOAD or CSAVE commands.

The CSAVE command saves a program on cassette tape. CSAVE takes one argument which can be any printing character. CSAVE can be given directly or in a program. Before giving the CSAVE command start your audio recorder on Record, noting the position of the tape.

CSAVE writes data on channel 7 and expects the device status from channel 6. Patches can easily be made to change these channel numbers.

When CSAVE is finished, execution will continue with the next statement. What is written onto the tape is BASIC's internal representation of the program in memory. The amount of data written onto the tape will be equal to the size of the program in memory plus seven.

Variable values are not saved on the tape, nor are they affected by the CSAVE command. The number of nulls being printed on your terminal at the start of each line has no affect on the CSAVE or CLOAD commands.

CLOAD takes its one character argument just like the CSAVE command. For example, CLOAD E.

The CLOAD command first executes a "NEW" command, erasing the current program and all variable values. The CLOAD command should be given before you put your cassette recorder on Play.

BASIC will read a byte from channel 7 whenever the character ready flag comes up on channel 6. When BASIC finds the program on the tape, it will read all characters received from the tape into memory until it finds three consecutive zeros which mark the end of the program. Then BASIC will return to command level and type "OK".

Statements given on the same line as a CLOAD command are ignored. The program on the cassette is not in a checksummed format, so the program must be checked to make sure it read in properly.

If BASIC does not return to command level and type "OK", it means that BASIC either never found a file with the right filename character, or that BASIC found the file but the file never ended with three consecutive zeros. By carefully watching the front panel lights, you can tell if BASIC ever finds a file with the right name.

Stopping the ALTAIR and restarting it at location 0 will prevent BASIC from searching forever. However, it is likely that there will either be no program in the machine, or a partial program that has errors. Typing NEW will always clear out whatever program is in the machine.

Reading and writing data from the cassette is done with the INP, OUT and WAIT statements. Any block of data written on the tape should have its beginning marked with a character. The main thing to be careful of is allowing your program to fall behind while data passes by unread.

Data read from the cassette should be stored in a matrix, since

there isn't time to process data as it is being read in. You will probably want to detect the end of data on the tape with a special character.

APPENDIX J   SLAAT OP 8080 VERSIE

BASIC/MACHINE LANGUAGE INTERFACE

In all versions of BASIC the user can link to a machine language subroutine. The first step is to set aside enough memory for the subroutine. When BASIC asks "MEMORY SIZE?", you shouldn't type a return, because BASIC would then write into all of memory trying to find out how much memory your machine has and then use whatever memory it finds.

The memory that BASIC actually uses is constantly modified, so you cannot store your machine language routine in those locations.

BASIC always uses memory starting at location 0 and as high upwards as you let it. BASIC cannot use non-contiguous blocks of memory. Therefore, it is best to reserve the top locations of memory for your machine language program.

For example, if you have a 4K machine and want to use a 200 byte subroutine, you should set memory size to 3896. Remember, BASIC always accepts numbers in decimal and that 4K is really 2↑12=4096 rather than 4000. Now BASIC will not use any location >= 3896.

If you try to allocate too much memory for your machine language program, you will get an OM (out of memory) error. This is because there is a certain amount of memory that BASIC must have or it will give an OM error and go back to the "MEMORY SIZE?" question.

The starting location of your routine must be stored in a location known as "USRLOC". The exact octal location of USRLOC will be given with each distributed version of BASIC. It is not the same for the 4K and 8K versions.

USRLOC for Version 3.0: 8K (both paper tape & cassette) = 111(octal)
4K = 103(octal)

Initially USRLOC is set up to contain the address of "ILLFUN", which is the routine that gives an FC (function call) error. USRLOC is the two byte absolute address of the location BASIC calls when USR is invoked.

USR is a function just like ABS or INT and is called as follows:
10 X=USR(3).

When your routine is called the stack pointer is set up and you are allowed to use up to 8 levels of stack (16 bytes). If you want to use more, you have to save BASIC's stack pointer (SP), set up your own, and restore BASIC's before you return back to BASIC.

All of the registers (A, B, C, D, E, H, L and PSW) can be changed. It is dangerous to modify locations in BASIC itself unless you know what you are doing. This is unlikely unless you have purchased a source copy of BASIC. Popping more entries off of the stack than you put on is almost guaranteed to cause trouble.

To retrieve the argument passed to USR, you must call the routine whose address is given in location 4 and 5 (DEINT). The low order 8 bits of an address are always stored in the lower address (4 in this case), and the high order 8 bits are stored in the next (higher) memory address (5 in this case).

Usr   poke 287 + 288 TO user progra.

The argument to USR is truncated to an integer (calling USR with 3.8 is the same as calling it with 3). If the argument is greater than 32767 or less than -32768, an FC error will result. When DEINT returns, the two byte signed value of the argument will be in registers D & E. The high order byte would be in D, the low order byte in E. For instance: if the argument to USR was -1, D would equal 255 and E would equal 255; if the argument was 400, D would equal 1 and E would equal 144.

To pass back a value from USR, set up a two byte value in registers A & B and call the routine whose address is given in locations 6 and 7. A & B should be set up in the same manner that D & E are when a value is passed to USR (A should contain the high order byte and B the low order byte).

If the routine whose address is given in locations 6 and 7 is not called, the function USR in the user's program will be an identity function. That is, USR(X) will equal X.

At the end of the USR routine a RET must be done to get back to BASIC. The BASIC program is completely stopped while USR is being executed and the program will not be continued until USR returns.

In the 4K version, the USR routine should not enable interrupts from a device. 4K BASIC uses the RST 7 location (56 decimal, 70 octal) to store a subroutine. If an interrupt occurs, this subroutine will be called which will have an undetermined and undesirable effect on the way BASIC behaves.

In the 8K BASIC, locations 56, 57 and 58 decimal have been set aside to store a JMP to a user-provided interrupt service routine. Initially a RET instruction is stored at location 56, so until a user sets up the call to his interrupt service routine, interrupts will have no effect.

Care must be taken in interrupt routines to save and restore the stack pointer, (A, B, C, D, E, H & L) and the PSW. Interrupt routines can pass data using PEEK, and can receive data using POKE.

The interrupt service routine should re-enable interrupts with an EI instruction before it returns, as interrupts are automatically disabled when the interrupt occurs. If this procedure is not followed, the interrupt service routine will never "see" another interrupt.

Though there is only one way of calling a machine language subroutine, this does not restrict the user to a single subroutine. The argument passed to USR can be used to determine which routine gets called. Multiple arguments to a machine language routine can be passed with POKE or through multiple calls to USR by the BASIC program.

The machine language routine can be loaded from paper tape or cassette before or after BASIC is loaded. The checksum loader, an unchecksummed loader, the console switches, or more conveniently the POKE function can be used to load the routine.

A common use of USR for 4K users will be doing IN's and OUT's to special devices. For example, on a 4K machine a user wants USR to pass back the value of the front panel switch register:

Answer to MEMORY SIZE? : 4050
USRLOC patched to contain [17,322]=7722 Base 8=4050 decimal

At location 4050=7722 Base 8 put:

```
7722/333     IN     255     ;(255 Base 10=377 Base 8) Get
7723/377                    ;the value of the switches in A
7724/107     MOV    B,A     ;B gets low part of answer
7725/257     XRA    A       ;A gets high part of answer
7726/052     LHLD   6       ;get address of routine
7727/006
7730/000                    ;that floats [A,B]
7731/351     PCHL           ;go to that routine which will
                            ;return to BASIC
                            ;with the answer
```

MORE ON PEEK AND POKE *(8K VERSION ONLY)*

As mentioned before, POKE can be used to set up your machine language routine in high memory. BASIC does not restrict which addresses you can POKE. Modifying USRLOC can be accomplished using two successive calls to POKE. Patches which a user wishes to include in his BASIC can also be made using POKE.

Using the PEEK function and OUT statement of 8K BASIC, the user can write a binary dump program in BASIC. Using INP and POKE it is possible to write a binary loader.

PEEK and POKE can be used to store byte oriented information. When you initialize BASIC, answer the MEMORY SIZE? question with the amount of memory in your ALTAIR minus the amount of memory you wish to use as storage for byte formatted data.

You are now free to use the memory in the top of memory in your ALTAIR as byte storage. See PEEK and POKE in the Reference Material for a further description of their parameters.

APPENDIX K

ASCII CHARACTER CODES

DECIMAL	CHAR.	DECIMAL	CHAR.	DECIMAL	CHAR.
000	NUL	043	+	086	V
001	SOH	044	,	087	W
002	STX	045	-	088	X
003	ETX	046	.	089	Y
004	EOT	047	/	090	Z
005	ENQ	048	0	091	[
006	ACK	049	1	092	\
007	BEL	050	2	093	]
008	BS	051	3	094	↑
009	HT	052	4	095	←
010	LF	053	5	096	`
011	VT	054	6	097	a
012	FF	055	7	098	b
013	CR	056	8	099	c
014	SO	057	9	100	d
015	SI	058	:	101	e
016	DLE	059	;	102	f
017	DC1	060	<	103	g
018	DC2	061	=	104	h
019	DC3	062	>	105	i
020	DC4	063	?	106	j
021	NAK	064	@	107	k
022	SYN	065	A	108	l
023	ETB	066	B	109	m
024	CAN	067	C	110	n
025	EM	068	D	111	o
026	SUB	069	E	112	p
027	ESCAPE	070	F	113	q
028	FS	071	G	114	r
029	GS	072	H	115	s
030	RS	073	I	116	t
031	US	074	J	117	u
032	SPACE	075	K	118	v
033	!	076	L	119	w
034	"	077	M	120	x
035	#	078	N	121	y
036	$	079	O	122	z
037	%	080	P	123	{
038	&	081	Q	124	|
039	'	082	R	125	}
040	(	083	S	126	~
041	)	084	T	127	DEL
042	*	085	U		

LF=Line Feed   FF=Form Feed   CR=Carriage Return   DEL=Rubout

CHR$ is a string function which returns a one character string which contains the ASCII equivalent of the argument, according to the conversion table on the preceding page. ASC takes the first character of a string and converts it to its ASCII decimal value.

One of the most common uses of CHR$ is to send a special character to the user's terminal. The most often used of these characters is the BEL (ASCII 7). Printing this character will cause a bell to ring on some terminals and a "beep" on many CRT's. This may be used as a preface to an error message, as a novelty, or just to wake up the user if he has fallen asleep. (Example: PRINT CHR$(7);)

A major use of special characters is on those CRT's that have cursor positioning and other special functions (such as turning on a hard copy printer).

As an example, try sending a form feed (CHR$(12)) to your CRT. On most CRT's this will usually cause the screen to erase and the cursor to "home" or move to the upper left corner.

Some CRT's give the user the capability of drawing graphs and curves in a special point-plotter mode. This feature may easily be taken advantage of through use of ALTAIR BASIC's CHR$ function.

APPENDIX L

EXTENDED BASIC

When EXTENDED BASIC is sent out, the BASIC manual will be updated to contain an extensive section about EXTENDED BASIC. Also, at this time the part of the manual relating to the 4K and 8K versions will be revised to correct any errors and explain more carefully the areas users are having trouble with. This section is here mainly to explain what EXTENDED BASIC will contain.

INTEGER VARIABLES These are stored as double byte signed quantities ranging from -32768 to +32767. They take up half as much space as normal variables and are about ten times as fast for arithmetic. They are denoted by using a percent sign (%) after the variable name. The user doesn't have to worry about conversion and can mix integers with other variable types in expressions. The speed improvement caused by using integers for loop variables, matrix indices, and as arguments to functions such as AND, OR or NOT will be substantial. An integer matrix of the same dimensions as a floating point matrix will require half as much memory.

DOUBLE-PRECISION Double-Precision variables are almost the opposite of integer variables, requiring twice as much space (8bytes per value) and taking 2 to 3 times as long to do arithmetic as single-precision variables. Double-Precision variables are denoted by using a number sign (#) after the variable name. They provide over 16 digits of accuracy. Functions like SIN, ATN and EXP will convert their arguments to single-precision, so the results of these functions will only be good to 6 digits. Negation, addition, subtraction, multiplication, division, comparision, input, output and conversion are the only routines that deal with Double-Precision values. Once again, formulas may freely mix Double-Precision values with other numeric values and conversion of the other values to Double-Precision will be done automatically.

PRINT USING Much like COBOL picture clauses or FORTRAN format statements, PRINT USING provides a BASIC user with complete control over his output format. The user can control how many digits of a number are printed, whether the number is printed in scientific notation and the placement of text in output. All of this can be done in the 8K version using string functions such as STR$ and MID$, but PRINT USING makes it much easier.

DISK I/O EXTENDED BASIC will come in two versions, disk and non-disk. There will only be a copying charge to switch from one to the other. With disk features, EXTENDED BASIC will allow the user to save and recall programs and data files from the ALTAIR FLOPPY DISK. Random access as well as sequential access will be provided. Simultaneous use of multiple data files will be allowed. Utilities will format new disks, delete files and print directories. These will be BASIC programs using special BASIC functions to get access to disk information such as file length, etc. User programs can also access these disk functions, enabling the user to write his own file access method or other special purpose

disk routine. The file format can be changed to allow the use of other (non-floppy) disks. This type of modification will be done by MITS under special arrangement.

OTHER FEATURES Other nice features which will be added are:

- Fancy Error Messages
- An ELSE clause in IF statements
- LIST, DELETE commands with line range as arguments
- Deleting Matrices in a program
- TRACE ON/OFF commands to monitor program flow
- EXCHANGE statement to switch variable values (this will speed up string sorts by at least a factor of two).
- Multi-Argument, user defined functions with string arguments and values allowed

Other features contemplated for future release are:

- A multiple user BASIC
- Explicit matrix manipulation
- Virtual matrices
- Statement modifiers
- Record I/O
- Paramaterized GOSUB
- Compilation
- Multiple USR functions
- "Chaining"

EXTENDED BASIC will use about 11K of memory for its own code (10K for the non-disk version) leaving 1K free on a 12K machine. It will take almost 20 minutes to load from paper tape, 7 minutes from cassette, and less than 5 seconds to load from disk.

We welcome any suggestions concerning current features or possible additions of extra features. Just send them to the ALTAIR SOFTWARE DEPARTMENT.

APPENDIX M

BASIC TEXTS

Below are a few of the many texts that may be helpful in learning BASIC.

1) BASIC PROGRAMMING, John G. Kemeny, Thomas E Kurtz, 1967, p145

2) BASIC, Albrecht, Finkel and Brown, 1973

3) A GUIDED TOUR OF COMPUTER PROGRAMMING IN BASIC, Thomas A Dwyer and Michael S. Kaufman; Boston: Houghton Mifflin Co., 1973

Books numbered 1 & 2 may be obtained from:

People's Computer Company
P.O. Box 310
Menlo Park, California
94025

They also have other books of interest, such as:

101 BASIC GAMES, Ed. David Ahl, 1974 p250

WHAT TO DO AFTER YOU HIT RETURN or PCC's FIRST BOOK OF COMPUTER GAMES

COMPUTER LIB & DREAM MACHINES, Theodore H. Nelson, 1974, p186